TECHNIQUES TO MITIGATE PERFORMANCE IMPACT OF OFF-CHIP DATA MIGRATIONS IN MODERN GPU COMPUTING

A Dissertation
Presented to
The Academic Faculty

By

Hyojong Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2020

TECHNIQUES TO MITIGATE PERFORMANCE IMPACT OF OFF-CHIP DATA MIGRATIONS IN MODERN GPU COMPUTING

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Moinuddin Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Date Approved: Jan 6, 2020

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Hyesoon Kim, from the bottom of my heart. Without her guidance and support throughout these years, this dissertation would not have been possible. I would also like to extend my deepest gratitude to my committee members, Dr. Ada Gavrilovska, Dr. Milos Prvulovic, Dr. Moinuddin Qureshi, and Dr. Vivek Sarkar for providing insightful and valuable feedback on my thesis. I would like to take this opportunity to extend my sincere thanks to the late Dr. Sudhakar Yalamanchili. His enthusiasm for the field was always inspiring and his kind support was very appreciated and he will be dearly missed. I also had great pleasure of working with Dr. Nuwan Jayasena while I was doing my research internship at AMD. I would like to acknowledge all the help I received from my collaborators: Dr. Gi-Ho Park, Dr. Arun F. Rodrigues, Dr. Yasuko Eckert, Dr. Onur Kayiran, Dr. Gabriel H. Loh, and Dr. Myoungsoo Jung. Many thanks to my labmates not just for research collaboration but for fun interactions: Minjang Kim, Sunpyo Hong, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jen-Cheng Huang, Joo Hwan Lee, Dilan Manatunga, Pranith Kumar, Prasun Gera, Nimit Nigania, Lifeng Nai, Ramyad Hadidi, Jiashen Cao, Farzon Lotfi, Yonghae Kim, Jaewon Lee, Bahar Asgari, and Blaise Tine. Thanks should also go to my friends: Joonseok, Sookyung, Edward, Jaegul, Hyung Suk, Kyu Yun, Taeheon, Hyemin, Hyoukjun, Jaehoon, Sungkap, Dongjin, Jiyeon, Haekyu, Woosang, Chaeyi, Seungyeon, Byoungyoung, Sunjae, Ching-Kai, Jongse, Ickhyun, Changhyun, Minsuk, and Jinwoo. I cannot leave Georgia Tech without mentioning Hannah. You are my best friend, rival, and support. My last thanks goes to my parents, Youngwoon Kim and Soonyoung Jung, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graphics processing units (GPUs) have been used successfully for accelerating a wide variety of applications over the last decade. In response to growing compute and memory capacity requirements, modern systems are equipped to distribute the work over multiple GPUs and pool the memory from the host (i.e., system memory) and other GPUs transparently. Compute capacity scales out with multiple GPUs, and the memory capacity afforded by the host is an order of magnitude larger than the GPUs' device memory. However, both these approaches require data to be migrated over the system interconnect (e.g., PCI-e) during program execution. Since migrating data over the system interconnect takes much longer than a GPU's internal memory hierarchy, the efficacy of these approaches in achieving high performance is strongly dependent on the data migration overhead. This dissertation proposes several techniques that help mitigate this data migration overhead.

In a system with multiple GPUs, where there is a large discrepancy in access times between local and remote memory accesses, it is crucial to co-locate compute and data to achieve high performance. This thesis discusses how to enable co-location of compute and data in such systems. The proposed mechanism estimates the amount of exclusive data and selectively allocates it in a single GPU while distributing the shared data across multiple GPUs. For this selective coarse-grained allocation, it uses a dual address mode with lightweight changes to virtual to physical page mappings. To place compute in the same GPU as the data it accesses, it uses an affinity-based thread block scheduling policy. This enables efficient use of multiple GPUs while minimizing unnecessary off-chip data migrations.

Support for unified virtual memory (UVM) and demand paging in modern GPUs provides a coherent view of a single virtual address space between CPUs and GPUs. This allows GPUs to access pages that reside in CPU memory as if they were local to the GPU. This enables GPU applications that are otherwise impossible to run due to memory capacity

constraints to run seamlessly. This thesis discusses how to alleviate major inefficiencies that arise in the page fault handling mechanism employed in contemporary GPUs. The proposed mechanism supports a CPU-like thread block context switching to reduce the number of batches (i.e., a group of page faults handled together) and amortize the batch processing overhead. To take page eviction off the critical path, it modifies the runtime software to overlap page evictions with CPU-to-GPU page migrations without requiring any hardware changes.

CHAPTER 1
INTRODUCTION

In response to unprecedented demand for compute and memory, modern graphics processing units (GPUs) allow use of multiple GPUs in a system or use of system memory (i.e., CPU memory) in a user-transparent manner. The use of multiple GPUs in a system scales out compute capability of the system, whereas the use of system memory provides an order of magnitude larger memory capacity to a GPU application. However, both techniques require data to be migrated over the system bus (e.g., PCIe bus in modern systems) on demand during execution. Provided that data migration over the PCIe bus takes much longer than what traditional GPUs are designed for, the efficacy of these techniques in provisioning high performance depends on mitigating the data migration overhead. This dissertation focuses on data migration challenges that each technique faces, and presents solutions.

1.1 Multi-GPU Systems: Challenges and Solutions

Challenges. In a system with multiple GPUs, there is a large discrepancy in access times between local and remote memory accesses. A local memory access is fast and efficient (exploiting large internal memory bandwidth), but a remote memory access is slow and inefficient (due to lower bandwidth remote interconnection network). Therefore, it is crucial to co-locate compute and data to achieve high performance.

However, two key techniques that have been used in traditional GPU systems to hide memory latency and improve thread-level parallelism (TLP), memory interleaving and thread block scheduling, are at odds with efficient use of multiple GPUs. Distributing data across multiple GPUs to improve overall memory bandwidth utilization incurs frequent data migration over the PCIe bus when the data and compute are misaligned. Nondeterministic thread block scheduling to improve compute resource utilization impedes co-placement of

compute and data.

Solutions. Our goal is to enable co-location of compute and data in the presence of fine-grain interleaved memory with a low-cost approach. To this end, we present a mechanism that identifies exclusively accessed data and places the data along with the thread block that accesses it in the same GPU. The key ideas are (1) the amount of data exclusively used by a thread block can be estimated, and that exclusive data (of any size) can be localized to one GPU with coarse-grain interleaved pages, (2) using an affinity-based thread block scheduling policy, we can co-place compute and data together, and (3) by using a dual address mode with lightweight changes to virtual to physical page mappings, we can selectively choose different interleaved memory pages for each data structure.

1.2 GPU Systems with Unified Memory Support: Challenges and Solutions

Challenges. To allow use of CPU memory in a user-transparent manner, modern GPUs support unified virtual memory (UVM) and demand paging. While these techniques substantially reduce programmer's burden on running large-scale GPU applications, they have a significant performance implication. We first investigate how current software runtime and hardware operates for UVM. The GPU runtime processes a group of GPU page faults together (batch processing) rather than processing each individual one, in order to amortize the overhead of multiple round-trip latencies over the PCIe bus and to avoid invoking multiple interrupt service routines in the operating system (OS). To efficiently process an excessive amount of page faults, the GPU runtime performs a series of operations such as preprocessing all the page faults and inserting page prefetching requests, which takes a significant amount of time (in the range of tens to hundreds of microseconds). Once all the operations (e.g., CPU page table walks for all the page faults, page allocation and eviction scheduling, etc.) are finished, page migrations between CPU and GPU begin.

Based on our in-depth analysis on how the GPU runtime handles GPU page faults, schedule page migrations, and interacts with the GPU hardware, we observe two major

inefficiencies that arise in the page fault handling mechanism employed in modern GPUs. First, the batched processing of page faults introduces a large scale serialization in page fault handling, greatly hurting GPU throughput. Second, page migrations between CPU and GPU are serialized and account for another significant fraction of page fault handling.

Solutions. Our goal is to mitigate these inefficiencies. We present a GPU runtime software and hardware holistic solution that (1) reduces the number of batch processing and amortizes the batch processing overhead by supporting CPU-like thread block context switching, and (2) takes page eviction off the critical path with no hardware changes by overlapping evictions with CPU-to-GPU page migrations.

1.3 Thesis Statement

Modern GPUs suffer from off-chip data migrations; lightweight software/hardware co-operative solutions that enable co-location of compute and data, and alleviate major inefficiencies that arise in the page fault handling mechanism employed in modern GPUs can improve the performance of modern GPU systems.

1.4 Thesis Outline

The thesis is organized as follows. Chapter 2 provides background on modern GPU computing: (1) an overview of how a multi-GPU system is organized and works, and (2) how unified memory and demand paging is supported by contemporary GPUs. Chapter 3 demonstrates the challenges in multi-GPU systems, and present solutions. Chapter 4 discusses preliminary studies on PCIe bus and page prefetching algorithms. It provides a better understanding on underlying techniques used in GPU systems with UVM. Chapter 5 demonstrates the challenges in GPU systems with UVM, and present solutions. Chapter 6 summarizes related works. Chapter 7 concludes dissertation.

CHAPTER 2

BACKGROUND

This chapter provides necessary backgrounds on modern GPU computing. First, we describe an overview of multi-GPU systems. Next, we explain how unified virtual memory and demand paging is supported by contemporary GPUs.

2.1 Multi-GPU Systems

2.1.1 Overview of a Multi-GPU System

Figure 2.1 shows a high-level diagram of a system with multiple GPUs and the details of a GPU. GPU uses the single-instruction multiple-thread (SIMT) execution [1, 2, 3] and we assume every GPU in the system can communicate with each other. While this assumption may not entirely hold true in modern multi-GPU systems, where not all GPUs can access each other (i.e., peer-to-peer access is only enabled between parts of them based on the topology), with this assumption, the proposed solution (Section 3) can be applicable to not only future multi-GPU systems that feature all-to-all communication, but also NUMA systems (irrespective of processor types) and multi-chip-module GPU systems [4].

Each GPU has streaming multiprocessors (SMs) and off-chip links for remote data accesses - to/from other GPUs and the CPU - and a crossbar network that connects SMs and its memory. The CPU launches a GPU kernel, and the runtime system partitions and distributes thread blocks across GPUs in the system. Up to the number of SMs $\times$ the number of thread blocks per SM can concurrently run in each GPU. There are two kinds of networks in such system: (1) a network among GPUs (denoted as **Remote**), and (2) a network that connects SMs in a GPU to their local memory (denoted as **Local**). Such multiple GPUs with remote and local memory accesses are common in near-data processing (NDP) [5, 6, 7].

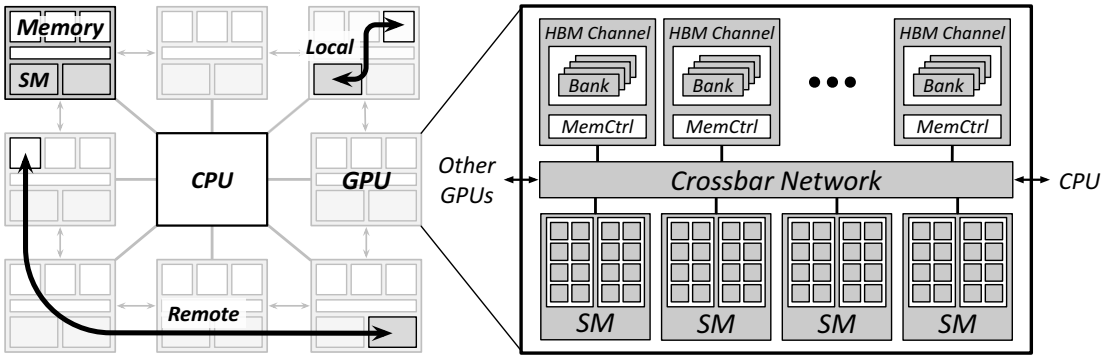


Figure 2.1: Overview of a system with multiple GPUs.

2.1.2 Address Interleaving

To increase memory-level parallelism, or to reduce channel/rank/bank conflicts, fine-grain memory interleaving is typically used in modern memory systems by striping small chunks of the physical address space (often the size of a few cache lines) across different banks, ranks, and channels. In a system with multiple GPUs, a page can be striped across multiple GPUs with fine-grain memory interleaving, or the entire page can be allocated in a single GPU with coarse-grain memory interleaving.

2.2 Unified Memory Support in GPUs

2.2.1 Thread Concurrency in GPUs

GPUs offer a high degree of thread-level parallelism (TLP) by executing thousands of scalar threads concurrently. To do so, the GPU shader core, such as NVIDIA Streaming Multiprocessor (SM), AMD Compute Unit (CU), or Intel Execution Unit (EU), provides hardware resources that are required to keep the contexts of multiple threads without doing conventional context switching. In each architecture, there are a number of factors that influence thread concurrency. For example, in NVIDIA GPUs, the maximum concurrency is capped by the maximum number of threads and thread blocks (e.g., 2048 and 32, respec-

tively), the register file size (e.g., 64k 32-bit registers), the maximum number of registers per thread (e.g., 256), among others. When a GPU kernel is launched, the GPU runtime decides the number of thread blocks to dispatch to each SM based on its hardware resources.

2.2.2 Unified Virtual Memory and Demand Paging

Modern GPUs offer unified virtual memory (UVM) that provides a coherent view of virtual memory address space among the CPUs and GPUs in the system [8, 9, 10]. For this, the GPU runtime software and hardware takes care of page migrations to the memory of the accessing processor under-the-hood. This eliminates the need for manual page migrations, which greatly reduces programmer's burden. Also, it enables the GPU applications that do not fit in the GPU memory to run. In this section, we describe how the unified memory works in detail.

Virtual Memory. Virtual memory support is required to allow any processor in the system to access the same data. The virtual-to-physical mapping is stored in a multi-level page table in GPUs. To translate a virtual address into a physical address, the GPU performs a page table walk. To accelerate this process, translation lookaside buffers (TLBs) are adopted from CPUs and optimized for GPUs [11, 12, 13, 14]. GPUs access an order of magnitude more number of pages than CPUs, requiring commensurate amount of translations. In light of this, a highly-threaded page table walker is proposed [12]. A multi-level page table requires many memory accesses to translate a single address. Exploiting the fact that the accesses to the upper level page table entries have significant temporal locality, a page walk cache [15] is also adopted for GPUs [12].

Demand Paging. When a GPU tries to access a physical memory page that is not currently residing in the GPU memory, the page table walk fails. The GPU generates a page fault and the GPU runtime migrates the requested page to the GPU memory. This page fault handling is expensive because (1) it requires long latency communications between CPU and GPU over the PCIe bus, and (2) the GPU runtime performs a very expensive fault

handling service routine. To amortize the overhead, the GPU runtime remedies a group of page faults together, which we refer to as batch processing.

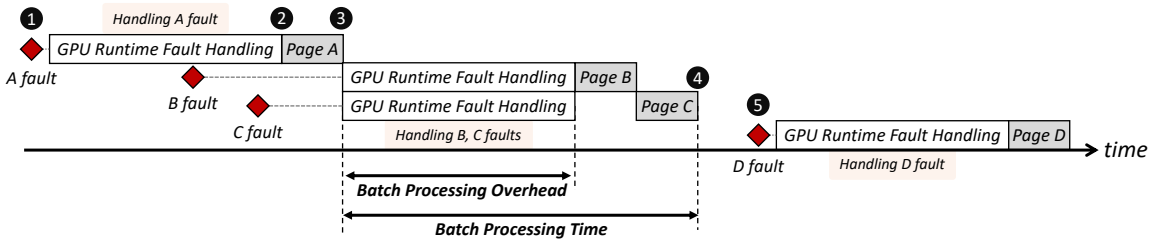


Figure 2.2: Overview of how GPU page faults are handled by the GPU runtime.

Figure 2.2 depicts an overview of how GPU page faults are handled by the GPU runtime. When a page fault exception is raised by the GPU memory management unit (MMU), the GPU runtime begins to handle the exception (❶). The exception handling starts by draining all of the page fault buffer entries (page A in the figure). We use one or two pages in the figure for simplicity, but in reality, a number of page faults are generated within a short period time, since thousands of threads are concurrently running under the SIMT execution model used in GPUs. To handle a plethora of page faults efficiently, the GPU runtime preprocesses the page faults before performing the page table walks. This preprocessing includes the sorting of the page faults in an ascending order of page addresses (to accelerate the page table walks) and the analysis of page addresses to insert page prefetching requests[1]. We refer to the time taken by the GPU runtime to perform a collection of operations to handle many page faults together as batch processing overhead. Specifically, the batch processing overhead is defined as the time between when a batch processing begins and when the GPU runtime begins to migrate the first page of the batch to/from the GPU memory. While the batch processing overhead is inherently non-deterministic (due to OS involvement), it also varies depending on the batch size (i.e., the number of page faults handled together in a batch) and contiguity of the pages.

The subsequent page faults generated after the batch processing begins (page B and C in

[1]Details on the preprocessing operations performed in a real GPU runtime can be found in `preprocess_fault_batch()` function in NVIDIA driver v396.37 [16].

the figure) cannot be handled along with page A. Instead, they are inserted into the page fault buffer and wait until current batch processing ends (❸). Once the page table walks are completed, the GPU runtime begins to migrate pages to the GPU memory (❷). Every time a page migration is done (❸), the GPU MMU updates its page table and resumes the threads that are waiting for the page. Once the last page migration is done (❸ and ❹), the batch processing ends. We refer to the time between when a batch processing begins and when the last page migration is done as batch processing time. When the batch processing ends, the GPU runtime checks if there are waiting page faults (page B and C in the figure). Then, the GPU runtime begins to handle them immediately. This is an optimization to reduce the unpredictable overhead that arises due to the interrupt-based service of the OS[2]. Otherwise, the GPU page fault processing routine ends (❹). This process is repeated when a new page fault interrupt is raised by the GPU (❺).

2.2.3 Case Study: Unified Virtual Memory in NVIDIA GPUs

In this section, we provide more detailed explanation on how to use UVM in NVIDIA GPUs, and the operations performed by the runtime system. UVM simplifies a lot of programming abstractions by presenting a unified memory space to the programmer. Allocating UVM is as simple as replacing calls to malloc() or new with calls to cudaMallocManaged(), an allocation function that returns a pointer accessible from any processor. When code running on a CPU or GPU accesses data allocated this way (i.e., CUDA managed data), the CUDA system software and/or hardware takes care of migrating memory pages to the memory of the accessing processor. On PASCAL and later GPUs, which support hardware page faulting and migration, CUDA managed data may not be physically allocated when cudaMallocManaged() returns; it may only be populated on access (or prefetching). In other words, pages and page table entries may not be created until they are accessed by the GPU or the CPU. The pages can be migrated to any processor's memory at any time, and the

[2]Details on this optimization performed in a real GPU runtime can be found in `uvm_gpu_service_replayable_faults()` function in NVIDIA driver v396.37 [16].

driver employs heuristics to maintain data locality and prevent excessive page faults[3]. The kernel launches without any migration overhead, and when it accesses any absent pages, the GPU stalls execution of the accessing threads, and the Page Migration Engine (PME) migrates the pages to the device before resuming the threads.

Starting from the PASCAL architecture, UVM functionality is significantly improved with 49-bit virtual addressing and on-demand page migration. 49-bit virtual addresses are sufficient to enable GPUs to access the entire system memory plus the memory of all GPUs in the system. The PME allows GPU threads to fault on non-resident memory accesses so the system can migrate pages on demand from anywhere in the system to the GPU's memory for efficient processing. In other words, UVM transparently enables oversubscribing GPU memory, enabling out-of-core computations for any code that is using UVM for allocations (e.g. cudaMallocManaged()). It does not require any modifications to the application. Also, system-wide atomic memory operations are supported. That means multiple entities (e.g., CPU and/or GPUs) can atomically operate on values anywhere in the system. This is useful in writing efficient multi-GPU co-operative algorithms.

Demand paging can be particularly beneficial to applications that access data with a sparse pattern. In some applications, it is not known ahead of time which specific memory addresses a particular processor will access. Without hardware page faulting, applications can only pre-load whole arrays, or suffer the cost of high-latency off-device accesses. With demand paging, only the pages the kernel accesses need to be migrated.

[3] Applications can guide the driver using cudaMemAdvise(), and explicitly migrate memory using cudaMemPrefetchAsync().

CHAPTER 3
ENABLING CO-LOCATION OF COMPUTE AND DATA FOR MULTI-GPU SYSTEMS

3.1 Introduction

In parallel programming models, such as the general-purpose graphics processing unit (GPGPU) programming model, the key to achieving high performance is to exploit thread-level parallelism (TLP). One way to accomplish this is to have each thread process a distinctive part of data such that the data process can be parallelized to the max. The effectiveness of this approach in achieving high performance critically depends on whether the system can hide memory latency and exploit all available compute resources. To hide memory latency and better exploit compute resources, modern GPU systems take two orthogonal approaches: memory interleaving and thread block scheduling, respectively. Memory interleaving is a technique that stripes small chunks of the physical address space across different memory modules, thereby increasing memory bandwidth utilization. Thread block scheduling determines to which GPU core each thread block is scheduled. Dispatching a thread block to an available GPU core in a round robin order would be the best way to provide load balancing, thereby, increasing resource utilization. While these techniques have been effective in traditional GPU systems, we question their efficacy in systems with multiple GPUs, since they might disrupt co-locating code and data.

Suppose a system has four GPUs, each with its own memory. GPUs are connected with the processor-centric topology [17], constituting the GPU memory address space. While a GPU can transparently access data in other GPUs, such an access uses the low bandwidth off-chip links and traverses the interconnect, incurring higher latency and leading to lower performance and energy efficiency. On the other hand, a local data access, which

occurs when a GPU accesses data in its local memory, utilizes high memory bandwidth, incurring lower latency and leading to higher performance and energy efficiency. The GPU physical address space is interleaved at a fine granularity to help improve memory bandwidth utilization. Let us take the transpose computation shown in Figure 3.1 as an example to examine the impact that memory interleaving and thread block scheduling have on performance. As its name suggests, this kernel transposes the `in` array and saves the result in the `out` array. Each thread processes distinctive `nfeatures` elements (line 4) from `(pid` $\times$ `nfeatures)-th` element of the `in` array (line 5).

```
1 __global__ void transpose(float *in, float *out, int npoints, int
    nfeatures) {
2   int pid = blockDim.x * blockIdx.x + threadIdx.x;
3   if (pid < npoints) {
4     for (int i = 0; i < nfeatures; i++)
5       out[i * npoints + pid] = in[pid * nfeatures + i];
6   }
7 }
```

Figure 3.1: Code snippet from K-means clustering.

Figure 3.2 and Figure 3.3 represent two cases where code and data are misaligned due to memory interleaving and thread block scheduling. For both of them, we assume interleaving granularity of 256 bytes, so four consecutive cache lines (cache line size is assumed to be 64 bytes) are placed in a GPU, and the next consecutive four cache lines are placed in the next GPU. We also assume the fair-round-robin thread block scheduling policy. A thread block is color-coded based on the GPU it is scheduled to, and a cache line is also color-coded based on which thread block accesses it. For example, thread blocks 0 and 4 have the same color, and lines 0, 1, 8, and 9 have the same color as thread blocks 0 and 4.

Figure 3.2 depicts a case in which each thread block processes two cache lines worth of elements of the `in` array. Note that the number of elements that each thread block processes is determined based on `nfeatures`. Accesses to lines 0 and 1 from thread block 0 are local, hence efficient, but accesses to lines 8 and 9 from thread block 4 are remote, and

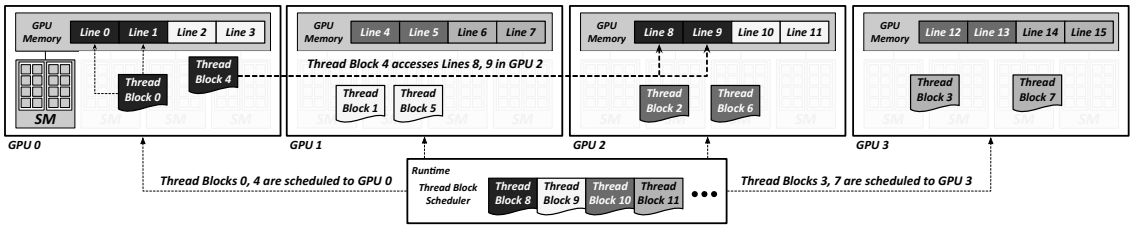


Figure 3.2: A case where code and data misalignment can be solved with thread block scheduling. Cache lines 8 and 9 are placed in GPU 2 due to memory interleaving. Thread block 4, which accesses them, is scheduled to GPU 0. This misalignment can be easily solved by scheduling thread block 4 to GPU 2.

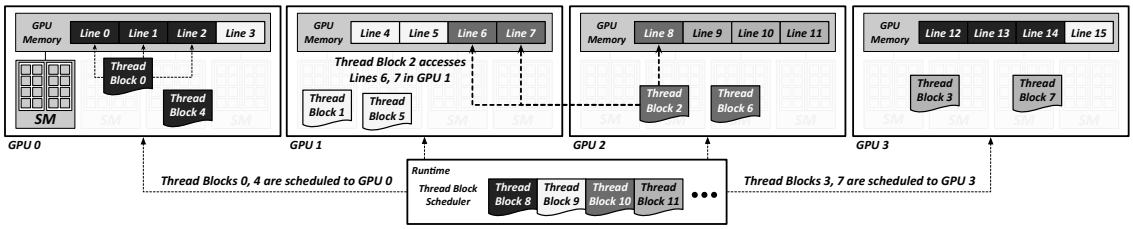


Figure 3.3: A case where code and data misalignment cannot be solved with just thread block scheduling. Cache lines 6 and 7 are placed in GPU 1, and cache line 8 is placed in GPU 2 due to memory interleaving. Thread block 2, which accesses them, is scheduled to GPU 2. Scheduling thread block 2 to GPU 1 makes accesses to cache line 8 inefficient.

hence inefficient. Fortunately, this misalignment can be easily solved by scheduling thread block 4 to GPU 2, where lines 8 and 9 are allocated. Figure 3.3 depicts a slightly different case in which each thread block processes three cache lines worth of elements of the `in` array. Now, it is not as simple as the case of Figure 3.2, since some of the accesses from a thread block are local and some are remote. Therefore, this misalignment cannot be solved just by scheduling a thread block to another GPU.

Our goal is to reduce such code and data misalignment, thereby achieving better performance. First, to place code and the data that it accesses together, we identify which data (and which part of it) each thread block accesses. We make two observations. First, the amount of data used by one thread block is often determined by the number of threads in a thread block and the amount of data each thread accesses. The latter can be estimated by either compile-time analyses (for input-independent access patterns) or profiler-assisted techniques

(for input-dependent access patterns). Second, although the number of threads in a thread block is often input-dependent, it is determined before kernel invocation (specifically, even before data structures are allocated). With these observations combined, we come to the conclusion that the amount of data used by one thread block can be estimated. For these reasons, we utilize a compiler-based and profiler-assisted technique to analyze the access pattern for each data structure and determine how each should be layered across GPUs.

Second, to place all the data that a thread block accesses in the same GPU as the thread block even in the presence of fine-grain memory interleaving, we make a slight change in hardware and the operating system (OS) to realize coarser-grain (the OS page size) memory interleaving in addition to the fine-grain (256 bytes) memory interleaving. The key idea is to use different sets of address mapping bits for each memory page depending on its anticipated access pattern, allowing the two sets of mappings to co-exist; low-order bits are used to distribute data across GPUs, whereas high order-bits are used to place an entire page in a single GPU. The granularity information for each memory page is stored in the page table entry (PTE) and translation lookaside buffer (TLB) entry. At the time a virtual address is translated into a physical address and the memory request is sent, our mechanism uses the appropriate address mapping depending on the granularity information. Admittedly, the concept of changing address mapping to change data layout or to increase memory-level parallelism is not new [18, 19]. However, our proposed mechanism is different from previous proposals in that it enables the coexistence of pages with different address mappings while not requiring large-scale page migrations.

Third, to ensure that the code is scheduled to the GPU where the data it accesses is located, we use an affinity-based scheduling mechanism. In traditional GPUs, thread blocks are scheduled to any GPUs (and any SMs in the GPU) in the system in a nondeterministic fashion. In order to steer a thread block and the data it accesses to the same GPU, we set an affinity between thread blocks and GPUs, and use the information for scheduling.

This work makes the following contributions:

- We observe that code and data alignment is critical in achieving high performance in a system with multiple GPUs, and traditional memory interleaving and thread block scheduling are at odds with efficient use of multiple GPUs.

- We propose a mechanism that utilizes a compiler-based and profiler-assisted technique to decide whether to localize or distribute each data structure based on its anticipated access pattern.

- We design a lightweight hardware mechanism that supports dual-mode address mapping at a page granularity, so that a page can be either spread across GPUs or localized to a single GPU. This mechanism enables pages with different address mappings to coexist in the same memory space and the amount of each mode can be adjusted at runtime.

3.2 Motivation

Figure 3.4 shows distribution of memory pages according to the number of thread blocks that access each memory page for various data-intensive workloads from publicly available GPU benchmark suites [20, 21, 22]. It is observed that for some workloads, such as `BFS`, `DC`, `PR`, `SSSP`, `BC`, `GC`, and `NW`, most pages are accessed by only one or two thread blocks. In traditional GPU systems, which have one GPU and its local memory, distributing pages irrespective of which and how many thread blocks access them helps improve the utilization of memory interfaces by distributing the memory traffic. However, in a system with multiple GPUs, where there is a big discrepancy between an access to local memory and that to remote memory, distributing such pages across GPUs incurs lots of remote traffics. Therefore, it is imperative to place such pages (exclusively used data) and the thread blocks (computations) that access them in individual GPUs. In contrast, in the case of `HS3D` and `HS`, most pages are accessed by almost all thread blocks. Even in the presence of multiple GPUs, it is better to distribute such pages (shared data) across GPUs to reduce memory bandwidth contention.

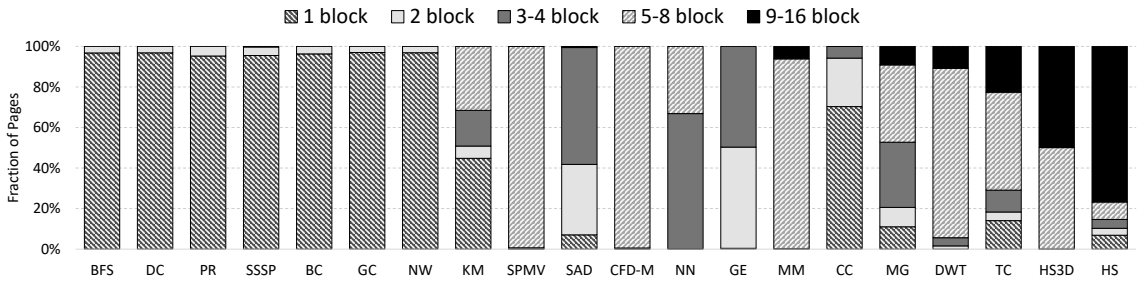


Figure 3.4: Distribution of memory pages according to the number of thread-blocks that access each page.

From this, we make two observations. First, some pages are accessed exclusively by a few thread blocks, while other pages are accessed, or shared, by many thread blocks. The exclusively used pages should be placed in individual GPUs with the thread blocks that access them to eliminate remote traffic, and the shared pages should be distributed across GPUs to reduce memory bandwidth contention. Second, each application has different distribution of exclusive and shared pages. For example, most pages in BFS are exclusively used, so the memory system should be capable of localizing all of them. On the other hand, most pages in HS are shared, so the memory system should also be capable of distributing all of them. These observations motivate the need for a mechanism that can allocate localized pages versus distributed pages *flexibly* based on an application's needs.

3.3 Mechanism

In this section, we describe our mechanism that enables co-location of compute and data in a system with multiple GPUs. Section 3.3.1 demonstrates how our mechanism can improve the case where code and data misalignment cannot be easily solved with just thread block scheduling, as shown in Figure 3.3. Section 3.3.2 describes a mechanism that either distributes data across GPUs or localizes data to a single GPU at a page granularity. Section 3.3.3 describes a mechanism that utilizes a compiler-based and profiler-assisted technique to decide whether to localize or distribute each memory page based on its anticipated access pattern and introduces an affinity-based scheduling algorithm that steers thread

blocks to the GPU where the data they access is located.

3.3.1 Demonstration

Figure 3.3 in Section 3.1 depicts a case where each thread block accesses three consecutive cache lines and thread blocks are scheduled with the fair-round-robin scheduling policy. In that example, due to the misalignment, just by improving the thread block scheduler cannot eliminate remote accesses. For example, accesses to cache lines 6 and 7 from thread block 2 are remote, hence inefficient, whereas accesses to cache line 8 is local, hence efficient. Scheduling thread block 2 to GPU 1 converts accesses to cache lines 6 and 7 to local, but the previous local access (access to cache line 8) becomes remote.

Figure 3.5 demonstrates how our mechanism solves this misalignment. First, our mechanism identifies which cache lines are accessed by which thread blocks (Section 3.3.3). Second, based on the identification and analyses, it decides whether to distribute the data across GPUs with fine-grain memory interleaving, or allocate them in a single GPU with coarse-grain memory interleaving. Our mechanism enables selective coarse-grain allocation on top of fine-grain interleaved memory (Section 3.3.2).

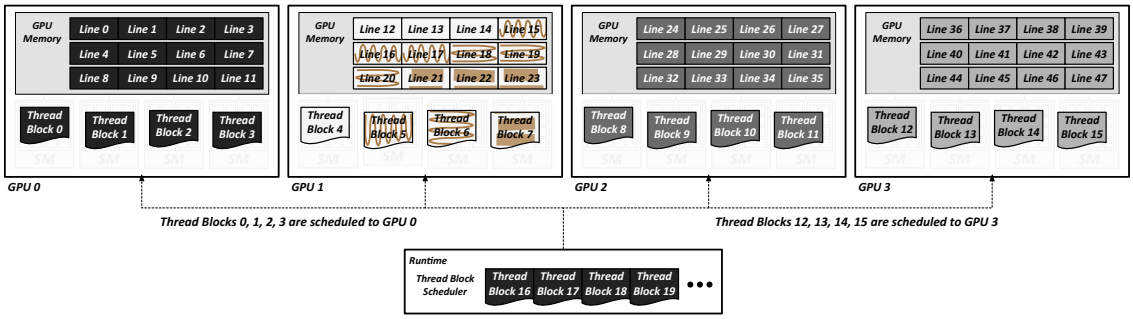


Figure 3.5: Representation of what our mechanism can do in the case where code and data misalignment cannot be easily solved with just thread block scheduling. It allocates consecutive cache lines 0-11 in GPU 0 and schedules thread blocks 0-3 to GPU 0 so that all the accesses to these cache lines will be efficient. Cache lines 12-23 are marked to represent which thread blocks access them.

3.3.2 Dual-mode Address Mapping

Hardware Support. To localize exclusively used pages in the presence of fine-grain memory interleaving, we use different sets of bits for address mapping for each memory page depending on the anticipated access patterns, allowing the two sets of mappings to co-exist. The default (fine-grain) address mapping distributes a page across GPUs (as is done today), and the alternative (coarse-grain) address mapping allocates (or localizes) an entire page in a single GPU (as is desirable for exclusively used data). We refer to the distributed page as FGP (fine-grain interleaved page) and the localized page as CGP (coarse-grain interleaved page). FGP is better suited for the data that is shared among (or accessed by) multiple GPUs. On the other hand, CGP is better suited for the data that is exclusively accessed by a single GPU. Note that once hardware provides the ability to map an entire page to a GPU (as is enabled by our selective use of coarse-grain address mapping), the OS could allocate arbitrarily large objects within a GPU by mapping all the virtual pages of that object to the physical pages (CGPs) in the GPU.

PTEs, TLB entries and cache lines are extended to indicate the granularity information, fine-grain or coarse-grain, for each page, as shown in Figure 3.6. The granularity bit in a PTE is set by the OS when a CGP is allocated, and the granularity bit in a cache line is set when the cache line is allocated. When the granularity bit is set, indicating CGP, the lowest bits from the PPN (Physical Page Number) are used to index GPU, whereas the highest bits from the page offset are used for FGPs. For example, in a system with four GPUs, when a cache line is evicted from the last level cache (LLC), a write-back request is sent to the memory indexed by either the bits [13:12] when the granularity bit is set (for CGPs) or the bits [11:10] when the granularity bit is not set (for FGPs). Be assured that we only change the mapping of the physical address to memory and not the physical address itself. Thus, cache is accessed with the original physical address, irrespective of the granularity information, and our mechanism does not have any impact on the cache coherence protocol or virtual address translation.

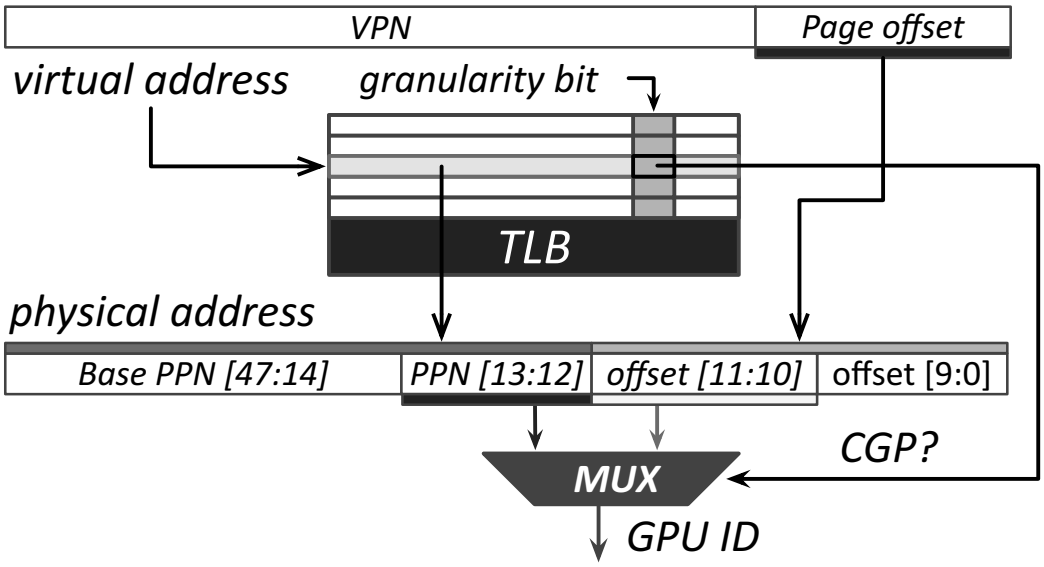
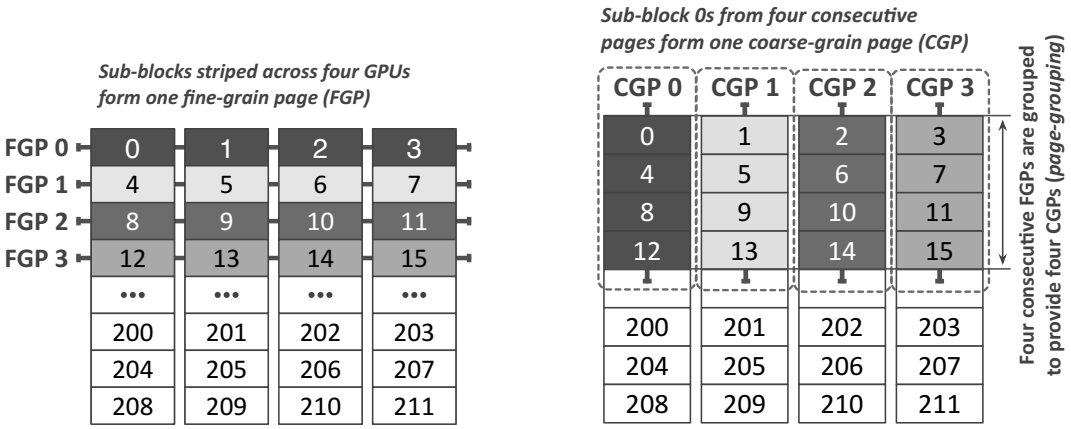


Figure 3.6: Hardware for a dual-mode address mapping.

System Software Support. The OS should be aware of the dual-mode address mapping (1) to indicate the granularity information in the PTEs and TLB entries, and (2) for page management, such as free page management or page replacement. It is important to note that it requires a set of adjacent FGPs to allocate a CGP (technically, a set of CGPs are allocated together). Consider a system where an FGP spans N consecutive GPUs, occupying a contiguous block of M bytes in each GPU memory. In that system, a CGP occupies N×M contiguous bytes within a single GPU memory. Therefore, a single CGP occupies the space that would have been utilized by N different FGPs within one GPU memory (but does not utilize any of the space those N FGPs would have occupied in other GPU memories). As a result, each block of N contiguous pages must uniformly be configured as FGP or CGP to avoid data layout conflicts. However, different blocks of N pages may be independently configured as FGP or CGP based on application or OS requirements.

For example, when FGP 0, in Figure 3.7 (a), consisting of sub-blocks 0, 1, 2 and 3, is converted to a CGP, there are conflicts with sub-blocks 4, 8 and 12 from the three subsequent

FGPs (each from FGP 1, FGP 2 and FGP 3, respectively). Therefore, those four FGPs must be converted to CGPs together, as shown in Figure 3.7 (b). We use the term *page-group* to refer to a set of pages that must be converted together. Hence, the OS should decide between FGP and CGP at a page-group granularity and can switch between FGP and CGP only when all the pages in the page-group are free.



(a) Fine-grain page layout

(b) Coarse-grain page layout

Figure 3.7: Conceptual diagram of page-group. The number indicates a sub-block address and the sub-blocks of the same color belong to the same OS page.

3.3.3 Compute-Data Co-location Algorithm

In traditional GPUs, thread blocks can be scheduled in any order, as they are supposed to run concurrently. The number of thread blocks that can run together in one SM is determined by thread block resource constraints. Normally, thread blocks are scheduled in order and as soon as one thread block retires, next thread block is scheduled to any available SM in any available GPU. However, to benefit from careful data placement, as is enabled by our dual-mode address mapping mechanism, thread blocks and the data they access must be co-located in the same GPU. To steer thread blocks and the data they access to the same GPU, we set an affinity between thread blocks and GPUs.

Affinity-based Work Scheduling Algorithm

We compute which GPU each thread block has affinity to using the following equation.

$$\text{affinity} = \left(\frac{\text{block\_id}}{N_{\text{blocks\_per\_GPU}}} \right) \bmod N_{\text{GPUs}} \tag{3.1}$$

block_id is flattened for multi-dimensional data based on row-major ordering (i.e., `blockIdx.y` $\times$ `blockDim.x` + `blockIdx.x`). $N_{\text{blocks\_per\_GPU}}$ is the number of thread blocks that can run concurrently in one GPU. For example, if one GPU has four SMs and each of which can run six thread blocks, $N_{\text{blocks\_per\_GPU}}$ is 24. When *N* is the number of GPUs and *T* is the total number of thread blocks, T/N thread blocks have the same affinity. With this affinity information, whenever an SM is available, instead of assigning any unscheduled thread block to it, the scheduler picks one that has affinity to that GPU.[1] This may potentially lead to load imbalance compared to the baseline of assigning any available thread block to any SM in the system. However, the number of thread blocks typically being much greater than the number of GPUs reduces the likelihood of load imbalance.

The hardware and runtime system must be extended to support this modified scheduling scheme. The scheduling algorithm could be optimized further to select thread blocks from other GPUs when a GPU does not have any work left to do, similar to the work-stealing algorithm. However, in our 20 evaluated benchmarks, only one suffered performance degradation due to the affinity-based scheduling algorithm. Therefore, we did not implement the work-stealing optimization.

Data Placement Algorithm

While the dual-mode address mapping enables the ability to localize an entire page in a single GPU, the question of how to identify the exclusively accessed or shared pages remains. This identification is particularly difficult for GPU systems because data structures are allocated

[1]This scheduling algorithm is conceptually similar to the guided scheduling policy in OpenMP, where programmer specifies chunk size (the number of loop iterations that one thread executes).

by the CPU before kernel invocations and are used by all threads in the kernel later.[2] To this end, we propose a compiler-based and profiler-assisted technique that identifies the amount of data used by one thread block for each data structure and decides which address mapping is desirable for the data structure (technically, for the pages in which the data structure is allocated). It is based on the following four observations. First, the amount of data used by one thread block is often determined by the number of threads in a thread block and the size of data structure that each thread accesses. Second, compile-time (symbolic) analyses can be used to detect if there exists a regular access pattern for each data structure. Third, profiler-assisted techniques can be used to estimate input-dependent accesses (more on this is explained later). Fourth, although the number of threads in a thread block is often input-dependent, it is determined before a kernel invocation (generally, even before data structures are allocated).

Based on these observations, we implement a compile-time analysis on LLVM infrastructure [23]. We extend the FunctionPass, which enables traversing all the kernel functions at compile time, and perform the symbolic analysis. For all the memory accesses inside kernels, we analyze the "GetElementPtrInst" LLVM instruction, which performs the index computation. Based on the index expression and the types of variables it uses, we examine if there exists a runtime-constant stride (RCS) between two consecutive thread blocks. In this examination, we check if an expression uses only the 1) kernel-invocation-constants, such as parameters, block/grid dimensions, or global constants, which are determined before kernel invocation and remain constant throughout the kernel execution, 2) thread index, thread block index, and/or loop index (for local loops in the kernel). If such a stride is found, we insert instructions in the CPU code to compute the stride distance between two consecutive thread blocks at runtime. We use profiler-assisted techniques for the case where the access pattern is input-dependent *and* only when the input is not changed frequently (e.g., graph computing workloads). Note that the profiler performs a similar examination as

[2]We only discuss global data structures, which may be accessed by all the threads in the system since local data structures are easily identifiable with specific keywords.

the compile-time analysis. Our mechanism also uses FGPs for irregularly accessed data, shared data, or parameter objects, as they are accessed by many thread blocks.

Algorithm 1 shows the compile-time analysis algorithm. We use the definition-use (DU) chains to trace back to the definition of all the source operands of each GetElementPtrInst LLVM instruction. To simply handle control flows within the IR, we consider only the initial value that reaches a PHI node, which is used by the LLVM IR to represent an SSA (static single assignment) form such that every use has exactly one reaching definition. That is, among the incoming values, we only consider the instruction that dominates the PHI node. For this purpose, we use the "DominatorTreeWrapperPass" analysis of LLVM. Here, for brevity, we only consider the case where index does not use value loaded from previous memory instruction. Also, we rule out the case where the RCS expression cannot be algebraically simplified to constants during compile-time. To handle these cases, we postpone the decision to runtime (before the kernel is launched) and instrument the code to compute the RCS for some random thread blocks. We take advantage of the fact that an accurate analysis is not necessary, since it is just used to decide the memory layout, not impacting correctness.

Where data should be located can also be computed, as the affinity-based work scheduling algorithm determines where computation will be performed. For example, if one thread block accesses the first B bytes of a data structure and N consecutive thread blocks will be scheduled to the SMs in a GPU, the mapping algorithm allocates contiguous chunks of $B \times N$ bytes on each GPU. The equations to compute chunk_size and stack_id are as follows:

$$\text{chunk\_size} = min(4\text{KB}, B \times N_{\text{blocks\_per\_GPU}}) \tag{3.2}$$

$$ID_{\text{GPU}} = \left(\frac{\text{virtual\_addr} - \text{obj\_start\_addr}}{\text{chunk\_size}} \right) \bmod N_{\text{GPUs}} \tag{3.3}$$

Please note that the chunk_size is upper-bounded by 4KB since an arbitrary number of pages can be allocated in a single GPU for any large object with hardware support to map an entire page to a single GPU with CGP. obj_start_addr is the starting virtual address of

Algorithm 1 Compile-time Runtime-Constant Stride (RCS) Analysis

Input: LLVM IR representation of GPU compute kernels

for each array index instruction : GetElementPtrInst **do**
  for each source operand **do**
    Recursively trace back to the root definition until *blockIdx* is found
    if A memory load is found **then** // uses value loaded from previous memory instruction
      Skip this instruction // *RCS* cannot be computed
    else if *blockIdx* is not found **then**
      Finish computation // *RCS* is zero
    else
      Continue
    end if
  end for
  if An *RCS* $\neq 0$ is found **then**
    Clone GetElementPtrInst to $index_\kappa$, and replace *blockIdx* with a constant K
    Create another clone of GetElementPtrInst to $index_{\kappa-1}$, and replace *blockIdx* with a constant $K-1$
    Create a subtract instruction $\Delta = index_\kappa - index_{\kappa-1}$, and perform algebraic simplification on it
    if Δ contains only kernel-invocation-constants, and the block and thread indices are canceled out **then**
      return RCS // Runtime-Constant Stride is found
    end if
  end if
end for

an object. When the chunk_size is not a multiple of physical page size, we round up to the next multiple of pages. The resulting misaligned pages will be shared by SMs from two consecutive GPUs, but this is still better than un-aligned distribution of data across all GPUs. Commonly, $N_{\text{blocks\_per\_GPU}}$ is moderately big since multiple thread blocks can run concurrently on an SM, which often results in a big chunk_size (greater or close to 4KB). Note that programs often use more than one data structure. Our proposed mechanism supports multiple data structures since we compute the chunk_size for each data structure using its own B size based on the structure's access pattern.

We demonstrate how our data placement algorithm works with Figure 3.1, a code snippet from K-means Clustering. The size of each data element can be identified and computed at compile-time, and the first element and the number of consecutive elements that each thread accesses can also be analyzed with our compile-time analysis routine. In this example, each thread accesses `nfeatures` consecutive elements from `(pid` $\times$ `nfeatures)-th` element, as shown in lines 4 and 5 of Figure 3.1. Since each thread block has `blockDim.x` threads, `blockDim.x` $\times$ `nfeatures` $\times$ `sizeof(float)` is the B value. This means that the first thread block accesses B bytes from the starting address of the `in` array and the second thread block accesses next B bytes. Note that the number of thread blocks and threads per thread block are determined before kernel invocation.

When a `cudaMalloc` function is called, our extended runtime system uses this information and the B value to compute the chunk_size using Equation (3.2) for the corresponding data structure and decides whether it should be allocated with the FGP or CGP. If a data structure is accessed by multiple kernels, the information of the first kernel that accesses it is used to compute the number of thread blocks per GPU. Accesses to 3D data structures are often more complicated than those to 1D or 2D data structures, for which the index is typically computed with both `blockDim.x` and `blockDim.y`. In this work, we focus on 2D data structures and leave the extension to support the 3D data structures and more complex data structures for the future work.

3.4 Evaluation

3.4.1 Methodology

Simulator. We evaluate our mechanism using SST [24] with MacSim [25], a cycle-level microarchitecture simulator. Low-level DRAM timing constraints are faithfully simulated using DRAMSim2 [26], which was modified to model the HBM 2.0 specification [27]. Our default system configuration comprises the CPU and four GPUs, where each GPU consists of four SMs and 8GB HBM memory. We model the GPU based on the NVIDIA Fermi architecture [28]. More details on the simulated system configuration are provided in Table 3.1. We use 128 bytes interleaving and 4KB interleaving to form the FGP and CGP, respectively. Each HBM channel is modeled to provide 32GB/s of peak memory bandwidth; therefore 256GB/s of total internal memory bandwidth is exploitable by each GPU. We model a Remote network to provide 16GB/s of memory bandwidth. We also perform detailed sensitivity studies, where we vary the bandwidth of Local and Remote networks.

Table 3.1: Configuration of simulated system

System	4 GPUs connected to the CPU with processor-centric topology [17]
GPU	Core: 4 2GHz SMs, fair-round-robin / affinity-based thread block scheduling policy Cache: 32KB core-private L1, 8-way, 4-cycle, 1MB shared L2, 16-way, 10-cycle Network: point-to-point network, 256GB/s Internal & 16GB/s Remote bandwidth
Memory	Each GPU has an 8GB HBM (HBM 2.0), interleaved at 128 bytes

Workloads. We use 20 memory-intensive benchmarks from GraphBIG [20], Rodinia [21], and Parboil [22]. We use the LLC MPKI (Misses Per 1000 Instructions) as an indicator for the memory-intensiveness. We classify a benchmark as being block-exclusive if almost all pages ($> 90\%$) are accessed by only one thread block, core-exclusive if almost all pages ($> 90\%$) are accessed by one GPU (i.e., multiple SMs in the same GPU), block-majority if the majority of pages ($> 60\%$) are accessed by only one thread block,

core-majority if the majority of pages ($> 60\%$) are accessed by one GPU, and sharing if most of the pages are accessed by multiple GPUs. Table 3.2 summarizes the benchmarks and the category they belong to.

Table 3.2: Benchmark categories

Category	Benchmarks
Block Exclusive	Breadth-First Search (BFS), Degree Centrality (DC), Page Rank (PR), Single-Source Shortest Path (SSSP), Betweenness Centrality (BC), Graph Coloring (GC), Needleman-Wunsch (NW)
Core Exclusive	K-means Clustering (KM), Gaussian Elimination (GE), k-Nearest Neighbors (NN), CFD Solver (CFD-M), Sparse-Matrix Dense-Vector Multiplication (SPMV), Sum of Absolute Differences (SAD), Dense Matrix-Matrix Multiply (MM)
Block Majority	Connected Component (CC)
Core Majority	MUMmerGPU (MG), Discrete Wavelet Transform (DWT)
Sharing	Triangle Count (TC), Hotspot3D (HS3D), Hybrid Sort (HS)

3.4.2 Performance

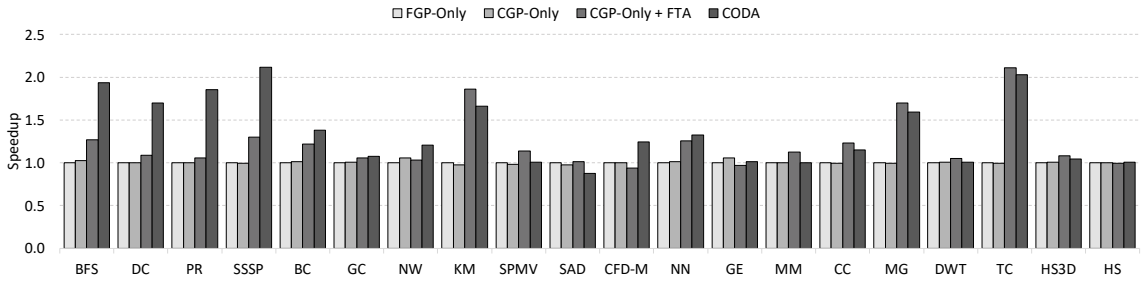


Figure 3.8: Speedup over FGP-Only, CGP-Only, and an ideal first-touch-based allocation scheme (CGP-Only + FTA).

Figure 3.8 shows the performance improvement of CODA for the benchmarks described in Table 3.2. FGP-Only represents the baseline where every page is interleaved at 128-bytes across GPUs, and CGP-Only represents the case where consecutive 4KB pages are allocated in consecutive GPUs in a circular order; this represents affinity-unaware data placement even when coarse-grain data allocation is available. CGP-Only+FTA (First-Touch-based

Allocation) represents the case where an entire page is allocated to the GPU that first touches the page. We ignore the accesses from the CPU in determining the first access, since all pages are initially allocated by the CPU before kernel invocation.[3] Even though this is not a practical implementation due to the lack of first-touch information at the time data is allocated (and often initialized) by the CPU, this can be a good indicator of the potential effectiveness of coarse-grain allocation for each benchmark. One simple way to implement first-touch-based allocation is to migrate pages upon first access. We observed that this migration-based first-touch allocation is not very effective (not shown, 7% speedup, as opposed to 19% speedup of CGP-Only+FTA) mainly due to small number of reuses of memory pages after migrations (due to burst and clustered access patterns); that is, the migration overhead is not mitigated. This makes a case for better data allocation rather than reactive data movement.

Our evaluation results show that CODA outperforms both FGP-Only and CGP-Only by 31%. CODA even outperforms CGP-Only+FTA for most benchmarks. For pages that are exclusively accessed by a single GPU, allocating those pages on that GPU brings a substantial reduction in remote data accesses and increase in local data accesses. This variation in remote and local data accesses directly leads to the performance improvement, as remote data accesses are limited by the low bandwidth of the off-chip links, whereas local data accesses exploit the large internal memory bandwidth. Perhaps more importantly, such bandwidth discrepancy becomes even more pronounced as the interconnection network becomes overwhelmed with more remote data accesses. Though lower bandwidth of the off-chip links does not necessarily mean longer memory access latency, when coupled with the off-chip communication overheads such as queuing delays and/or external transfer time, average memory access latency can be significantly affected by the number of remote data

[3]NVIDIA PASCAL architecture or later GPUs support demand paging and runtime page migration [8, 9]. On those devices along with a newly introduced data allocation API (`cudaMallocManaged`), the statement that all pages are initially allocated by the CPU may not be true. Pages can be allocated by a GPU, generating page faults. While conceptually uncomplicated to use, demand paging and page migration cause significant performance degradation [29, 13]. Therefore, in this study, we assume and evaluate traditional and more general GPU programming model.

accesses as well.

Notably, our mechanism localizes accesses *whenever possible* even for the benchmarks classified as sharing, in which most pages are accessed by many SMs (in multiple GPUs), thereby achieving performance improvements. The amount of performance gain each benchmark obtains depends on the distribution of accesses to page types (exclusive pages vs. shared pages). Specifically, if a majority of accesses are made to exclusive pages, the benchmark could gain a significant performance improvement from CODA. This is the case for TC, for example.

Overall, our mechanism achieves 1.56x and 1.13x average performance improvements over the baseline for block-exclusive and core-exclusive benchmarks, respectively. This is particularly effective in graph algorithms with large numbers of neighbor accesses (e.g., BFS, DC, PR, and SSSP), which are difficult to handle efficiently.

3.4.3 Local vs Remote Access

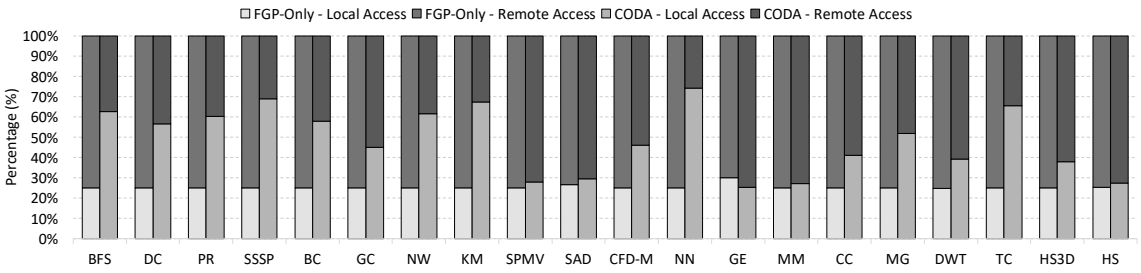


Figure 3.9: Comparison of local and remote data accesses between FGP-Only and CODA.

Figure 3.9 shows distribution of memory accesses, local versus remote, for the baseline and how it varies with our mechanism. Our mechanism significantly reduces remote data accesses for all the evaluated benchmarks but one, GE.[4] A substantial reduction in remote data accesses and an increase in local data accesses contribute to the performance

[4] As opposed to the case of TC in Section 3.4.2, GE has a majority of accesses to shared pages, and for this reason, remote data accesses are not reduced a lot with CODA, even though it is classified as core-exclusive because we classified benchmarks based on the distribution of pages, not based on the distribution of accesses to page types (exclusive page vs. shared page).

improvement for the following reasons. First, local data accesses can utilize the large internal memory bandwidth, while remote data accesses are limited by the lower memory bandwidth of the off-chip links. Second, for the remote data accesses, a great amount of time could be spent on waiting for network due to the off-chip communication. This can be incurred as a result of limited network bandwidth, but can be exacerbated further due to the artifacts of the off-chip communication, such as queuing delays, routing delays, etc. Our mechanism significantly reduces remote data accesses, enabling the utilization of large internal memory bandwidth and also mitigating the effect of interconnection network congestion by placing memory pages in the same GPU in which the computation is to be performed.

Our mechanism is especially effective for the block-exclusive and core-exclusive benchmarks. On average, 47% and 34% remote data accesses are reduced, respectively. Even for the sharing benchmarks, by identifying the pages that are accessed by a few thread blocks or SMs, and allocating them where the computation is to be performed, our mechanism reduces 32% remote data accesses.

3.4.4 Sensitivity to Bandwidth

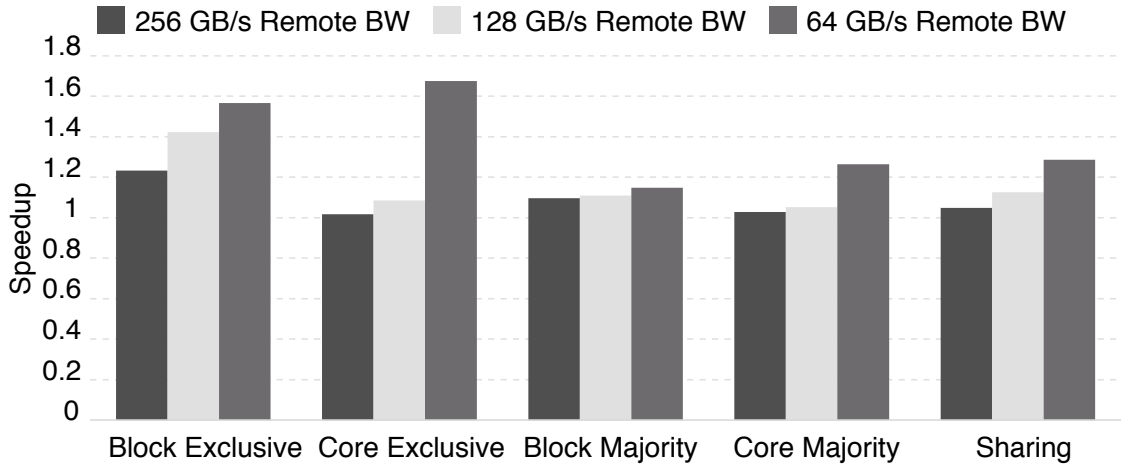


Figure 3.10: Speedup with different remote bandwidth among GPUs.

Even for highly provisioned systems with unrealistically large Remote bandwidth and low remote memory access latency, co-location of thread blocks and the data they access

improves performance, as shown in Figure 3.10. This is because even in such systems, remote memory accesses cannot be completely free from all resource conflicts. Careful data placement, as is enabled by our mechanism, can significantly reduce the possibility of such conflicts and therefore can contribute to the performance improvement.

This evaluation is to show how sensitive performance is to Remote bandwidth. We can observe that when 256GB/s of Remote bandwidth is available, Remote bandwidth is no longer a performance bottleneck (relative performances of all categories are close to 1, except for the block-exclusive category). The big gap in the core-exclusive category appears because the applications in that category are limited by Remote bandwidth of 64 GB/s and not by 128 GB/s Remote bandwidth.

Even when a system has 256 GB/s of aggregated Remote bandwidth, our mechanism improves performance by 8% (up to 23%). It should be noticed that as the gap between Local bandwidth and Remote bandwidth increases (Remote bandwidth is decreased while Local bandwidth remains the same), our mechanism provides more benefit by reducing remote data accesses and opening up more opportunity to exploit large internal memory bandwidth, thereby mitigating the performance penalty of the off-chip communication (performance improvement goes up to 15.2% and 37.4%, respectively).

3.4.5 Sensitivity to Graph Properties

In graph computing, the number of vertices and their neighbors that each thread block accesses highly depends on graph properties. To examine the impact of the graph properties on our proposed mechanism, we differentiate the properties that can be estimated at the time the graph is preprocessed[5] from those that cannot be estimated. Basic graph properties such as the number of vertices and edges can be obtained at the time the graph is preprocessed. These, combined with the number of threads per thread block, which is determined based

[5]The term preprocessing generally implies a heavy-weight operation such as a clever partitioning to reduce communication. In this study, however, we only extract basic properties of a graph without scanning through the entire graph.

on the resource constraints of the underlying hardware, can be used to estimate the average number of edges that each thread block accesses (μ) before kernel invocation and the standard deviation (σ) of it. The coefficient of variation of a graph, which can be estimated as σ/μ, is a good indicator of how regular a graph is: a graph with a small coefficient of variation is considered regular. Therefore, the granularity at which the graph should be distributed, or the block stride distance, can be determined.

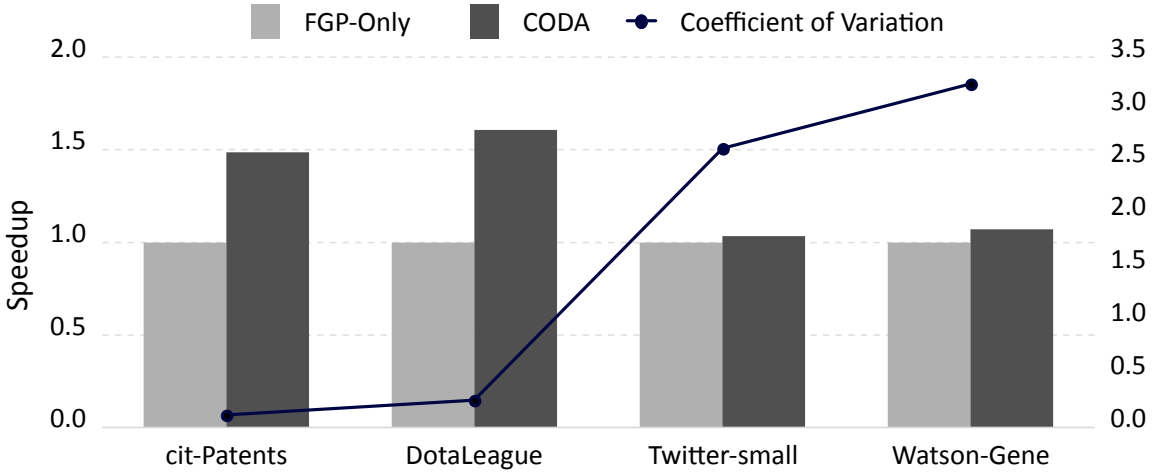


Figure 3.11: PageRank performance with different graphs

Figure 3.11 compares the performance of FGP-Only and CODA, using the PageRank workload. The evaluation is based on four real-world graphs, which have 59K to 9M vertices. Graphs are sorted based on their regularity: graphs with a smaller coefficient of variation appear toward the left side of the figure. The coefficient of variation of each graph is also depicted. We confirm that the effectiveness of our mechanism depends highly on graph properties. Regular graphs benefit more from our mechanism (55%) than irregular graphs (5%) since the estimation accuracy depends only on the properties that can be estimated at the time graph is preprocessed. Notably, CODA does *not* degrade performance in any case since it detects the memory pages that are exclusively accessed by one GPU and localizes them with CGP, while distributing other memory pages with FGP, as in the case of FGP-Only.

3.4.6 Multiprogrammed Workloads

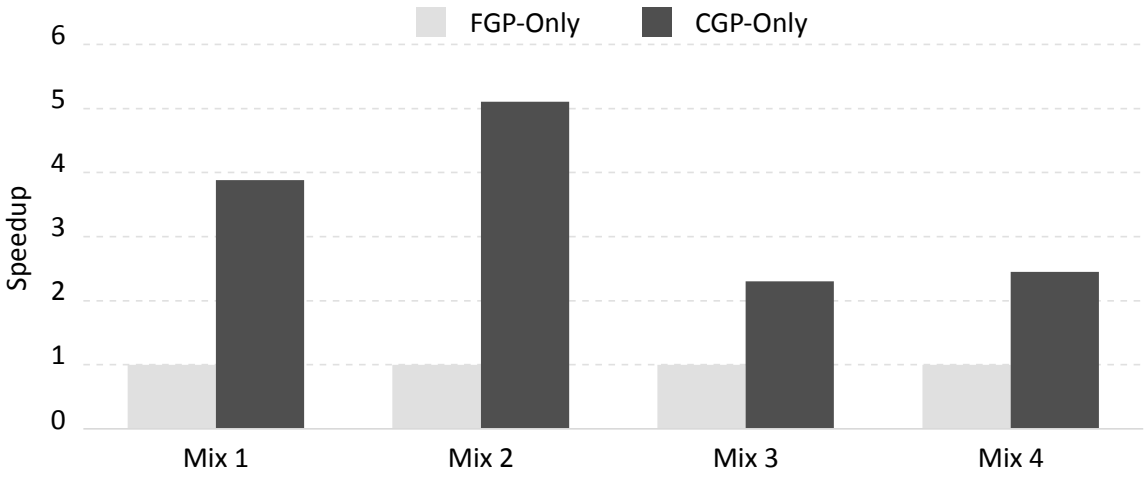


Figure 3.12: Performance of multiple applications

To further analyze the impact of having hardware that provides the ability to map an entire page to a single GPU using CGP, we evaluate our CGP-Only configuration with four mixes of multiprogrammed workloads. Each benchmark is chosen randomly from each category to construct a multiprogrammed workload. Figure 3.12 compares the performance of CGP-Only with that of FGP-Only, showing that the CGP-Only outperforms the FGP-Only for all the workloads. With FGP-Only hardware, every memory page is distributed across all GPUs, which results in a significant number of remote data accesses from all applications. With hardware that can map an entire page to a single GPU, as enabled by our mechanism, however, memory pages that an application accesses can be allocated to the GPU where the application is executed, and hence, all the accesses can exploit the large internal memory bandwidth within the GPU. This is an important contribution since it is infeasible or difficult to reduce remote data accesses in the presence of multiple workloads running in a system.[6]

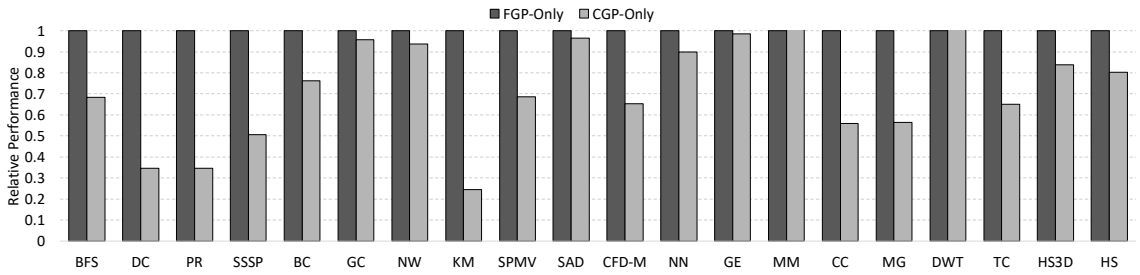


Figure 3.13: Performance impact of interleaving granularity

3.4.7 Impact of Interleaving Granularity

So far we have demonstrated the necessity of the coarse-grain memory interleaving (technically, selective use of CGP and FGP) for the efficient use of multiple GPUs. One might consider using just coarse-grain memory interleaving in a system with multiple GPUs. However, in this section we present the performance of FGP-Only and CGP-Only with a centralized GPU that has the same overall compute capability as that of all the GPUs in the multiple GPU system to demonstrate the necessity of the fine-grain memory interleaving as well. When an application runs on the centralized GPU, as in traditional GPU systems, it is desirable that the memory objects it accesses are distributed across multiple memories to achieve maximum memory bandwidth utilization by distributing concurrent accesses across all available memory interfaces. Figure 3.13 shows the performance of the GPU with memories interleaved at different granularities. FGP-Only and CGP-Only indicate the use of fine-grained interleaved memory and coarse-grained interleaved memory, respectively. Our evaluation results show that FGP-Only outperforms the CGP-Only by 1.48x due to better memory bandwidth utilization.

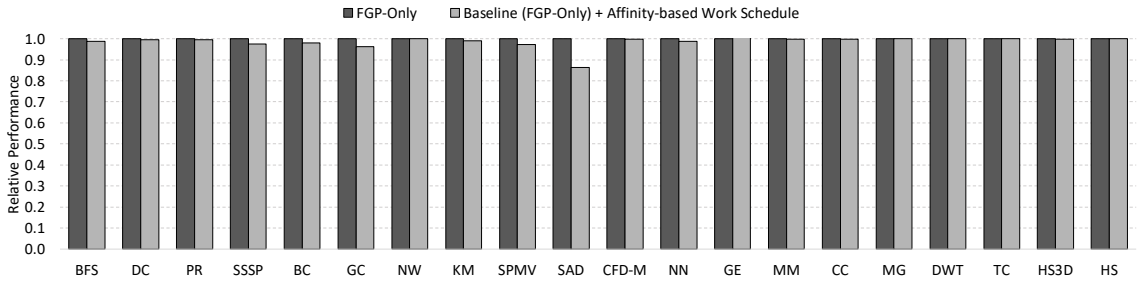


Figure 3.14: Performance impact of an affinity-based work scheduling mechanism

3.4.8 Impact of Affinity-based Scheduling

Thread blocks cannot be scheduled to any SM with our affinity-based work scheduling mechanism. In this section, we evaluate the performance impact of the affinity-based work scheduling mechanism. Figure 3.14 compares the performance of the affinity-based work scheduling mechanism (FGP-Only + Affinity-based Work Schedule) and that of the baseline (FGP-Only). All our evaluated benchmarks are virtually unaffected by the restricted scheduling mechanism, as expected, except for one benchmark, `SAD`. The reason why the performance of `SAD` is degraded by the affinity-based work scheduling is that the number of thread blocks is relatively small (61) considering the number of GPUs and available SMs (16). Maintaining load balancing across all available compute resources might be more crucial than carefully co-locating thread blocks and the data they access, when compute resource bounds the overall performance. This problem can be alleviated with resource-monitoring-based schemes.

3.5 Discussions

3.5.1 Complex Address Mapping

So far, we have assumed a simple address mapping scheme for ease of explanation. Modern processors, however, use more complex address mapping schemes such as XORing multiple

[6]Please note that this experiment is intended to show the necessity of having hardware that provides the ability to map an entire page to a single GPU, not to compare the performance of the baseline configuration (FGP-Only) and CODA, although it can be easily expected that CODA would perform as well as CGP-Only.

bits (not necessarily consecutive) for channel selection [30]. In this section, we discuss the applicability of our dual-mode address mapping mechanism in such systems. Note that computation and data co-location algorithm presented in Section 3.3.3 is orthogonal to the address mapping scheme used in the underlying system. Although the detailed address mapping scheme differs for different architectures, the mappings can be classified into those that use the channel-selection bits exclusively (i.e., they are not used as part of the row- or column-selection), and those that do not (i.e., at least one bit from the channel-selection bits is used as part of the row- or column-selection). Our dual-mode address mapping mechanism can be easily extended to support a system with the former class of mappings, where channel-selection bits are used exclusively, by swapping the channel-selection bits with other higher order bits after XOR operation. However, it is not trivial to support a system with the latter class of mappings, where channel-selection bits are not exclusively used. One way might be to identify which bits are used exclusively for the channel-selection and which bits are not, and then carefully swapping the channel-selection bits with those that are not used for channel-selection. This requires further investigation and is a part of our future work.

3.5.2 Large Page and Memory Management

Large pages have been used to mitigate address translation overheads by reducing the number of PTEs to maintain and increasing TLB hit rates. However, it comes at a cost, such as internal fragmentation, memory bloat, and increased load-to-use latency [13]. In this section, we discuss the applicability of our dual-mode address mapping mechanism for the large pages. Again, computation and data co-location algorithm presented in Section 3.3.3 is orthogonal to the page size. First, our dual-mode address mapping can be easily extended for the large pages. For 2MB pages, for example, address bits [22:21] can be used (instead of address bits [13:12] in the case of 4KB page) to index GPUs to allocate the entire page in a single GPU. However, the key challenge in supporting large page is not about choosing

which bits to use for GPU selection but about dealing with fragmentation issues. Although our mechanism may complicate page management and potentially increase fragmentation issues, we believe that if page-groups are small (e.g., 4 or 8 pages), this is likely to not be significantly more complicated than normal page management. Also, the memory manager can be modified to deal with page-groups for most operations (e.g., flushing out to disk) for better memory management. This requires further exploration and is a part of our future work.

3.5.3 PTE Extension

Our proposed mechanism requires a modification to the PTE format. X86 ISA reserves 3 bits [11:9] for future usage [31], so we can use one of the bits to indicate the granularity information. When a system employs large pages, extra bits are available in the PTE, which gives more freedom to modify PTE contents.

3.5.4 NUMA or NUCA Systems

In this section, we discuss the difference and uniqueness of our system from the conventional NUMA (Non-Uniform Memory Architectures) [32] or NUCA (Non-Uniform Cache Access) [33] systems in CPUs. First, in NUMA systems, memory policies such as node-local or interleave can be specified and (relatively) easily controlled. For example, the first-touch based page allocation has already been used in NUMA systems. On the contrary, the first-touch based page allocation *cannot* be used in GPU systems (GPUs prior to NVIDIA Pascal architecture [8]) due to the lack of first-touch information (recall that data structures are allocated and initialized by the CPU before kernel invocation). Even if the first-touch information *were* available, a memory page could not be allocated in a single GPU without hardware support for the localization. Furthermore, since shared data across multiple cores are often cached in the CPUs, the penalty of NUMA is often reduced. However, in GPUs, the cache size is much smaller than CPUs, so caches cannot hide the penalty of NUMA

easily. Second, NUCA systems (e.g., R-NUCA [33]) rely on data migration after an access pattern is identified. The migration overhead is much smaller in NUCA systems than in our multiple GPU system because the former migrates data within a single device (e.g., a tiled L2 cache architecture), whereas the latter migrates data across multiple devices connected via comparatively low-bandwidth, high-latency interconnect links.

CHAPTER 4

STUDIES ON TECHNIQUES EMPLOYED IN MODERN GPU SYSTEMS

4.1 PCIe

4.1.1 Introduction

A significant part of performance degradation from using UVM can be attributed to inefficient use of the PCIe bus. So as to alleviate the problem, understanding where the inefficiency arises is in order. Figure 4.1 shows PCIe latency (bar chart) and Speedup (line chart) of transferring various data sizes over the PCIe bus. Speedup is computed as the ratio of time taken to transfer 2MB of data with a given transfer size to time taken to transfer 2MB of data with 4KB transfers. The speedup increases as we increase the transfer size, and reaches a plateau region from around 64KB. This indicates that it takes 12x more time to transfer 2MB of data using 4KB transfers than using 64KB transfers or higher. From this, we observe that the PCIe transfer is inefficient when the transfer size is smaller than 64KB.

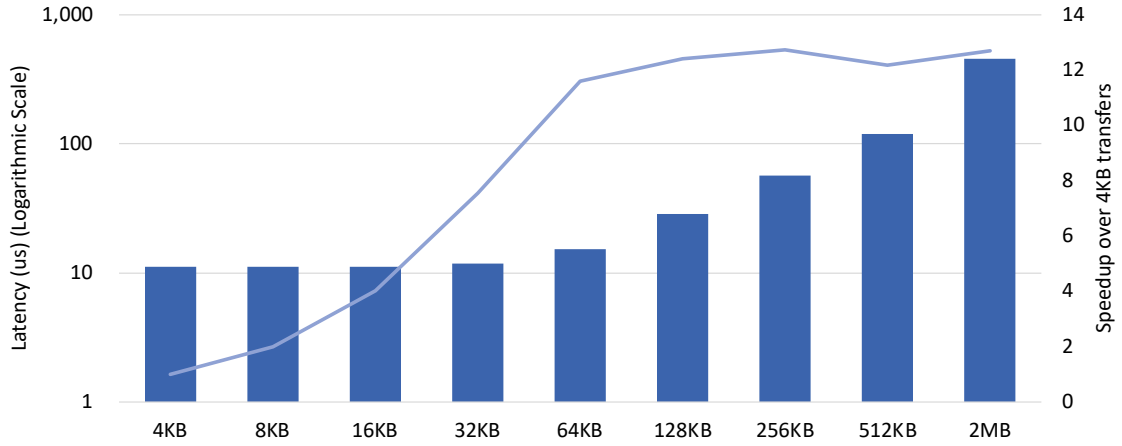


Figure 4.1: PCIe latency (bar chart) and efficiency (line chart) for various transfer sizes.

Figure 4.2 shows a breakdown of a PCIe transfer of various transfer sizes. Setup time represents PCIe initialization overhead and it is constant regardless of the transfer size.

Transfer time represents actual time to transfer data and it increases as the transfer size increases. We observe that the setup time is dominant up to transfer size of 16KB and that it becomes insignificant when the transfer size is big enough, e.g., 512KB or 2MB. Transferring distinct 4KB pages takes a toll due to this setup time overhead. Since the PCIe does not support a congregate transfer of non-contiguous regions, the performance degradation of managing GPU memory at a fine granularity, e.g., 4KB or 8KB, can be attributed to this overhead.

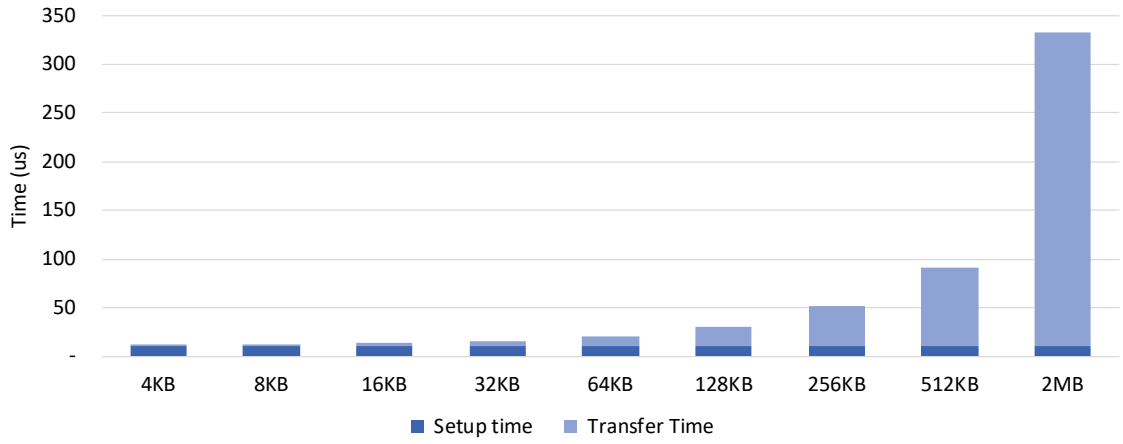


Figure 4.2: Breakdown of a single PCIe transfer overhead.

4.1.2 How to Use PCIe Efficiently?

We saw that the PCIe initialization overhead accounts for a dominant part of the total transfer time when transfer size is small. Increasing the transfer size can amortize the initialization overhead. However, caution should be exercised, since it takes more time to transfer more data. Therefore, a proper transfer size has to be picked in order to balance the initialization cost and the data transfer time. Note that page eviction time is on the critical path, even though the PCIe specification [34] allows bidirectional transfer. Therefore, a proper eviction size has to be picked as well in order to minimize the PCIe overhead and reduce the overall transfer time. While our proposed mechanism (Section 5.3.2) can take page eviction latency off the critical path, to not lose generality, we assume such mechanism is not used in this

study.

Performance impact of increasing page size has been studied a lot. However, imprudent use of large pages for GPU may cause significant slowdown. While using large pages reduces the address translation overhead (by reducing TLB miss rates), it is not necessarily suitable for demand paging. Demand paging for large pages requires a great amount of data to be transferred over the PCIe bus during a page fault, hurting performance due to the faulting threads being stalled longer [13]. Therefore, the ideal GPU memory management scheme would use small page for demand paging (but not too small in order to avoid inefficient use of PCIe) and use large pages for eviction (but not too large in order to prevent early eviction of live small pages). NVIDIA's memory management scheme for UVM confirms our findings. In NVIDIA's mechanism, when a page fault occurs, a large page frame (2MB) is reserved but not populated, and only a subset of the large page is transferred from CPU memory to GPU memory. By only transferring a subset of the entire large page, they minimize the load-to-use latency (i.e., the time between when a thread issues a load request and when the data is returned to the thread) [13]. Upon eviction, an entire large page (i.e., all constituent small pages of the large page) is evicted.

4.1.3 Design Space Exploration

In this section, we explore the performance impact of fetch size (denoted as Fetch-Subregion-Size) and evict size (denoted as Evict-Region-Size) to find a sweet spot for each application. There are pros and cons of using different sizes for fetch and eviction[1]. A bigger Fetch-Subregion-Size inherently provides the benefit of prefetching, thereby reducing GPU page faults. Also, as long as it is less than 64KB, the PCIe transfer latency is kept nearly constant. However, it may hurt the performance for applications with poor locality. A bigger Evict-Region-Size benefits from more efficient use of PCIe, but it may also hurt performance since the eviction delays the pending page transfers from CPU to GPU memory and it may

[1]We use the term Subregion for fetch size and Region for evict size because we always fetch subregions and always evict regions, which is bigger than subregions.

increase the likelihood of early evictions of live subregions, which have to be brought back again to the GPU memory in the future. In this section, we explore various combinations of FetchSubregion-Size and Evict-Region-Size for each application.

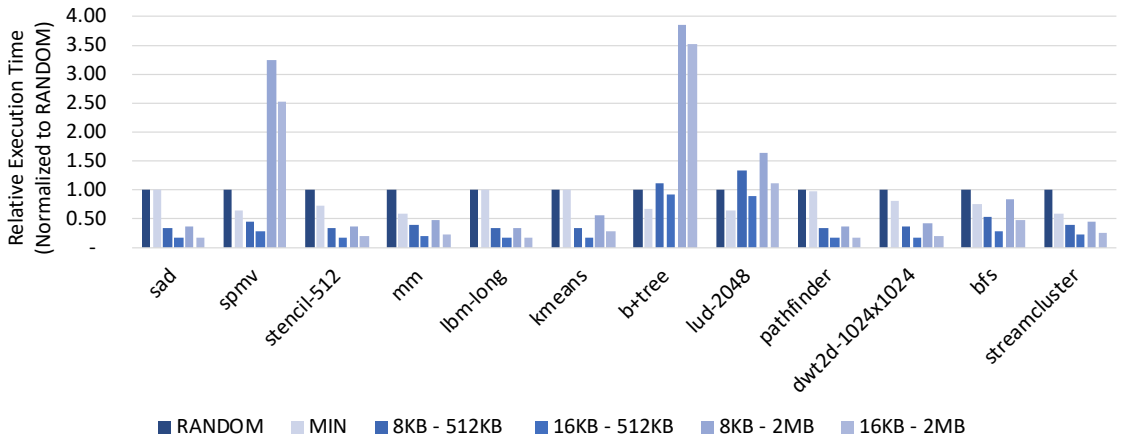


Figure 4.3: Performance impact of Fetch-Subregion-Size and Evict-Region-Size for regular workloads.

Figure 4.3 shows performance of various sets of Fetch-Subregion-Size and Evict-Region-Size for regular workloads. We use workloads that have footprints exceeding 32MB from the Rodinia [21] and Parboil [22] benchmark suites. Many applications benefit a lot from using large Evict-Region-Size, even compared to impractical MIN (Belady's algorithm) page replacement algorithm [35, 36, 37]. We evaluate MIN algorithm by running the application twice, first, to collect the entire history of page accesses, and second, to use the history to decide which page to evict. Across all the evaluated workloads, using 512KB Evict-Region-Size with 16KB Fetch-Subregion-Size performs the best. As expected, increasing the Fetch-Subregion-Size is beneficial in reducing GPU page faults, thereby improving overall performance. Applications like `SPMV` or `b+tree`, which have extremely poor locality, especially suffer from increasing Evict-Region-Size from 512KB to 2MB due to increased early evictions.

Figure 4.4 shows performance of various sets of Fetch-Subregion-Size and Evict-Region-Size for irregular workloads. We use workloads from the GraphBIG [20] benchmark suite.

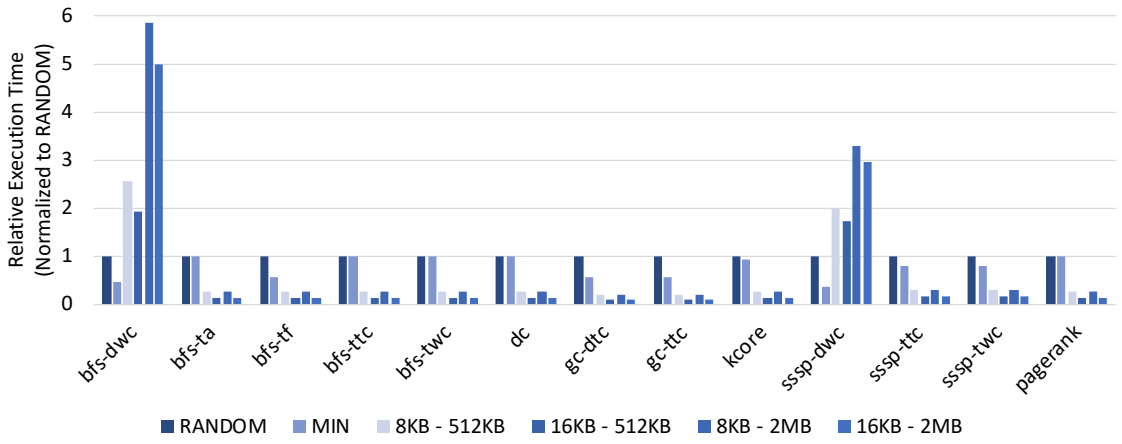


Figure 4.4: Performance impact of Fetch-Subregion-Size and Evict-Region-Size for irregular workloads.

Surprisingly, large Evict-Region-Size (e.g., 512KB or 2MB) outperform the best performing fine-grain memory management scheme (MIN) even for many irregular graph computing workloads. The exceptions are data-centric implementations of graph computing workloads. Since they exhibit orders of magnitude more irregularity, using large EvictRegion-Size to alleviate the performance loss due to the PCIe transfer exacerbates overall performance. These extremely irregular applications typically have very poor locality, so they favor more sophisticated fine-grain memory management schemes.

4.2 Page Prefetching

4.2.1 Introduction

Page prefetching has been studied in the context of modern GPUs [29, 38]. This section discusses performance impact of page prefetching by comparing various prefetching algorithms. To reduce complexity, we assume that GPU memory is unlimited (i.e., GPU memory oversubscription does not occur). We consider the following three page prefetching algorithms: Random, Sequential, and Tree-based Neighborhood. Random makes page selections by choosing pages randomly within a range of 2MB that the faulty page belongs to. Sequential chooses the rest of 64KB basic block that the faulty page belongs to (assum-

ing GPU memory is managed at a 64KB granularity). Tree-based neighborhood chooses the prefetch candidates in a rather adaptive manner [38]. In the tree-based neighborhood mechanism, GPU memory is managed as follows: each 2MB address space is divided into 4KB pages and consecutive 16 pages (64KB) are assigned into a leaf node, indicating that each 2MB space is represented by a single tree that has 32 leaf nodes. When a page fault occurs, the prefetcher marks the corresponding leaf node and selects all pages in the node as prefetch candidates. Then it moves up to the parent node and checks whether its marked child nodes are more than half. If so, it marks all of it child nodes and puts the corresponding pages into the prefetch candidate set. Otherwise, it does not mark any node at this point and just moves up to the parent node. This process is repeated until the root node of the 2MB region is reached. With this algorithm, a single page fault can initiate prefetching of as small as 60KB worth of pages and up to 1020KB worth of pages.

4.2.2 Performance Impact of Prefetching Algorithms

To understand the performance impact of various prefetching algorithms, we extend the MacSim [25] simulator. The PCIe bandwidth utilization varies depending on the transfer size [38]. To be pragmatic, we use the reported PCIe bandwidth: 3.2219 GB/s for 4KB transfer size, 6.4437 GB/s for 16KB transfer size, 8.4471 GB/s for 64KB transfer size, 10.508 GB/s for 256KB transfer size, and 11.223 GB/s for 1024KB transfer size. To be conservative, we use lower-level bandwidth (e.g., 6.4437 GB/s for the size between 16KB and 64KB) for interpolation. We select 18 out of 23 applications from the Rodinia [21] benchmark suite. The rest 5 applications are excluded due to impractically long simulation time.

Random, sequential, and tree-based neighborhood are denoted as rand, seq, and tree, respectively. For brevity, we show only 5 (backprop, bfs, hotspot, nw, and pathfinder) out of 18 evaluated applications. We use the entire results from the 18 applications to calculate average values. Figure 4.5 shows the number of page faults for each application when

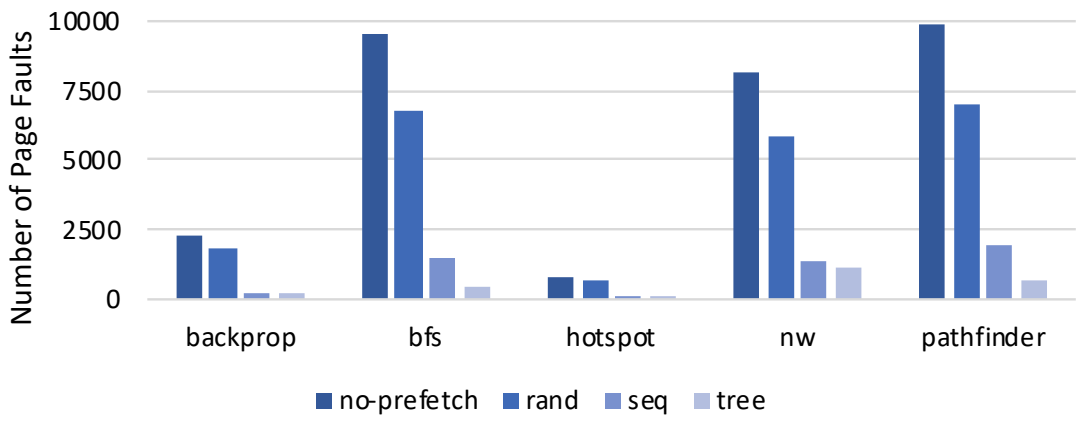


Figure 4.5: Number of GPU page faults.

each prefetch algorithm is used, and shows that the number of page faults is significantly reduced when the tree-based prefetcher is used. This is because the tree-based prefetcher prefetches pages more aggressively compared to others, and because we are assuming unlimited memory, where no page evictions occur, there is little downside to aggressive prefetching. While no-prefetch, random, and sequential prefetchers prefetch at most 0, 1, and 15 pages, respectively, for each demand page fault, the tree-based prefetcher can prefetch up to 255 pages depending on the marked states of the corresponding tree (2MB region). Compared to no-prefetch, random, sequential, and tree-based prefetcher reduce 20.75%, 69.15%, and 82.86%, respectively, of page faults on average across all 18 evaluated applications.

The decrease in the number of page faults is also reflected in the GPU page fault handling time, as shown in Figure 4.6. Compared to no-prefetch, random, sequential, and tree-based prefetcher reduce 5.35%, 53.35%, 75.77%, respectively, of page fault handling time on average across all 18 evaluated applications.

However, as shown in Figure 4.7, the total execution time does not change as dramatically as the page fault handling time. It is true that in most cases the tree-based prefetching mechanism shows the best performance, the gaps between other mechanisms are insignificant. Compared to no-prefetch, random, sequential, and tree-based prefetcher reduce 3.70%,

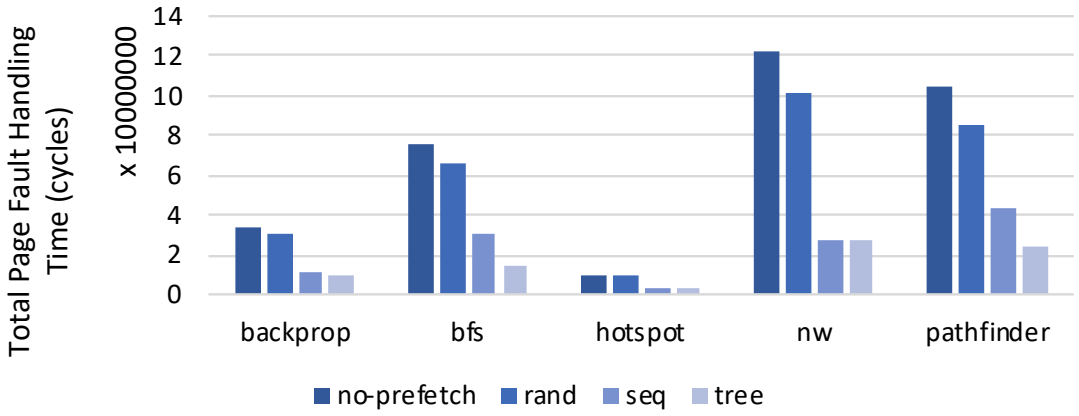


Figure 4.6: Time spent to handle GPU page faults.

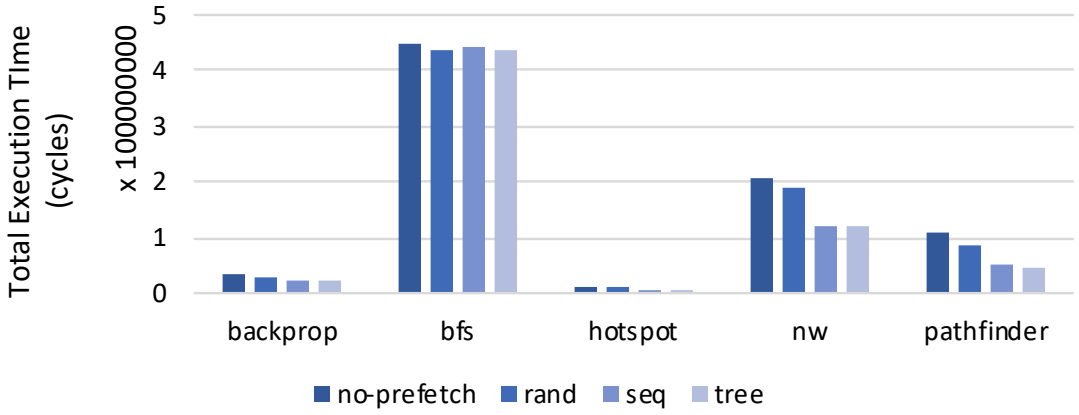


Figure 4.7: Total execution time.

13.14%, 16.65%, respectively, of total execution time on average across all 18 evaluated applications. The reason is two-fold. First, the proportion of the page fault handling time varies depending on applications. For instance, the proportion of the time spent for page fault handling to the total execution time is less than 17% for all mechanisms and it is only 2% with the tree-based mechanism. Based on Amdahl's law, it is reasonable that the reduction in page fault handling time affects the total execution time in a limited fashion.

Second, the evaluated applications have small working set size. For example, bfs has the largest memory footprint, which is 40MB. Considering that a single tree of tree-based prefetcher represents 2MB, the footprint is too small to correctly reflect the impact of the

mechanisms. To evaluate the prefetching mechanisms properly, it would be desirable to use applications with larger memory footprint or working set. We leave this for future work.

CHAPTER 5
BATCH-AWARE UNIFIED MEMORY MANAGEMENT IN GPUS FOR IRREGULAR WORKLOADS

5.1 Introduction

Graphics processing units (GPUs) have been successful in providing substantial compute performance and now become one of the major computing platforms in the servers and datacenters [39, 40]. As an accelerator device, however, conventional discrete GPUs only allow access to its own device memory, so programmers need to design their applications carefully to fit in the limited capacity of the device memory. This makes it very challenging and costly to run large-scale applications with hundreds of GBs of memory footprint, such as graph computing workloads, because it requires careful data and algorithm partitioning in addition to purchasing more GPUs just for memory capacity.

To address this issue, recent GPUs support Unified Virtual Memory (UVM) [8, 9, 10]. UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging. This allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory, thereby allowing GPU applications to run without worrying about the device memory capacity limit. As such, UVM frees programmers from tuning an application for each individual GPU and makes it run on a variety of GPUs with different physical memory sizes without any source code changes. This is good for programmability and portability.

While it all sounds promising, in reality, the benefit comes with a significant performance cost. Virtual memory support requires address translation for every memory request, and its performance impact is more substantial than in CPUs in general because GPUs can issue a significantly large number of memory requests at a cycle [12, 13]. In addition, paging in

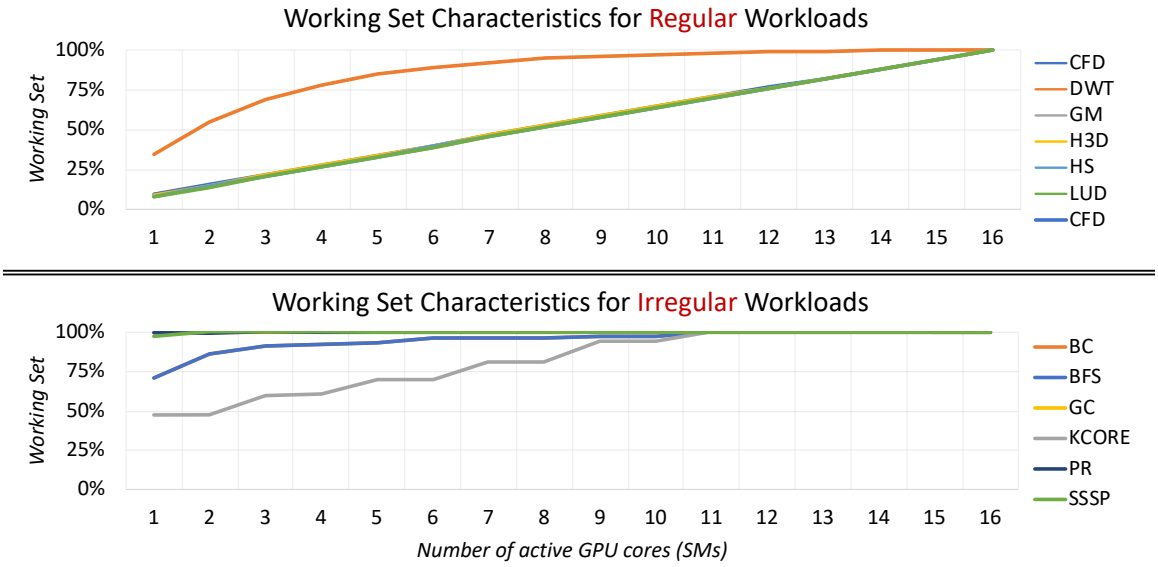


Figure 5.1: Working set vs. GPU core. For most of regular workloads, working set size is proportional to the number of active GPU cores. In many large-scale, irregular applications, however, most memory pages are shared across GPU cores, so GPU core throttling is ineffective in reducing working set size.

and out of GPU memory requires costly communications between CPU and GPU over the PCIe [41] and an interrupt handler invocation. Prior work reports that page fault handling latency ranges from 20μs to 50μs [29]. We find that these numbers are conservative and can be worse depending on the applications and systems. Unfortunately, this microsecond scale of page fault latency cannot be easily hidden even with ample thread-level parallelism (TLP) in GPUs, especially when GPU memory is oversubscribed.

Recently, Li et al. [42] have proposed eviction-throttling-compression (ETC), a memory management framework to improve GPU performance under memory oversubscription. Depending on the application characteristics, the framework selectively employs the proactive eviction, memory-aware core throttling (i.e., enabling/disabling GPU cores), and capacity compression techniques. However, for many large-scale, irregular applications, we find that the ETC framework is ineffective. First, the proactive eviction heavily relies on predicting the correct timing to avoid both early and late evictions. Since irregular applications access a large number of pages within a short period of time, predicting correct timing is not trivial.

Second, the memory-aware throttling technique aims to reduce the application working set by disabling GPU cores. In order for this to be effective, the working set has to be reduced when GPU cores are throttled. This is the case for most regular workloads, as shown in Figure 5.1. However, this is not the case for many large-scale, irregular applications because most of the memory pages are shared across GPU cores, and thus, memory-aware throttling is not effective in reducing the working set.

The goal of this work is to support efficient execution of large-scale irregular applications such as graph computing workloads in the UVM model. To be pragmatic, we first investigate how the current software runtime and hardware operates for UVM (Section 2.2.2). The GPU runtime processes a group of GPU page faults together rather than processing each individual one, in order to amortize the overhead of multiple round-trip latencies over the PCIe bus and to avoid invoking multiple interrupt service routines in the operating system (OS). To efficiently process an excessive amount of page faults, the GPU runtime performs a series of operations such as preprocessing all the page faults and inserting page prefetching requests, which takes a significant amount of time (in the range of tens to hundreds of microseconds). Once all the operations (e.g., CPU page table walks for all the page faults, page allocation and eviction scheduling, etc.) are finished, page migrations between CPU and GPU begin. Section 5.2 describes our findings in detail along with the assessment of processing a group of page faults in a real GPU system.

Based on our in-depth analysis on how the GPU runtime handles GPU page faults, schedule page migrations, and interacts with the GPU hardware, we observe two major inefficiencies that arise in the page fault handling mechanism employed in modern GPUs. First, the batched processing of page faults introduces a large scale serialization in page fault handling, greatly hurting GPU throughput. Second, page migrations between CPU and GPU are serialized and account for another significant fraction of page fault handling. To mitigate these inefficiencies, we present a GPU runtime software and hardware holistic solution. First, it reduces the number of batch processing and amortizes the batch processing overhead

by supporting CPU-like thread block context switching. Second, it takes page eviction off the critical path with no hardware changes by overlapping evictions with CPU-to-GPU page migrations.

The key contributions of this paper are as follows:

- This is the first to discuss that a group of page faults are handled together in a batch in contemporary GPUs. We provide a comprehensive analysis on major inefficiencies that arise from this batch processing mechanism.
- We provide an insight that when page migrations occur and account for a significant portion of the execution time, we have to consider dispatching more thread blocks to each GPU core even at the cost of context switching.
- We demonstrate that these inefficiencies can be alleviated with lightweight and practical solutions, thereby enabling very low cost demand paging for GPUs.
- We improve performance by 2x and 1.81x over the state-of-the-art page prefetching mechanism [29] and ETC mechanism [42], respectively.

5.2 Motivation

We make two observations on the batch processing mechanism. First, batch processing introduces a significantly large scale delay to subsequent page fault groups. Take the page B fault in Figure 2.2 as an example, which is generated in the GPU after the first batch processing begins. Since it cannot be handled along with page A, it has to wait until the current batch processing ends. To provide perspective, we profile a BFS application on a real NVIDIA Titan Xp [8]. The batch processing time is measured to be in the range of 223μs and 553μs with a median of 313μs, of which, the batch processing overhead accounts for an average of 46.69% (measured to be in the range of 50μs and 430μs with a median of 140μs).

There are fundamentally two ways to mitigate the impact of this delay. The first is to

reduce the batch processing overhead itself by optimizing the GPU runtime software, and this is beyond the scope of our work. The second is to amortize the batch processing overhead. This can be attained by increasing the batch size (i.e., the number of page faults handled together in a batch). Figure 5.2 shows that per-page batch processing overhead decreases as the batch size increases (from the aforementioned profiling). In Section 5.3.1, we discuss the reason why it is challenging to increase the batch size, and propose a technique to achieve it.

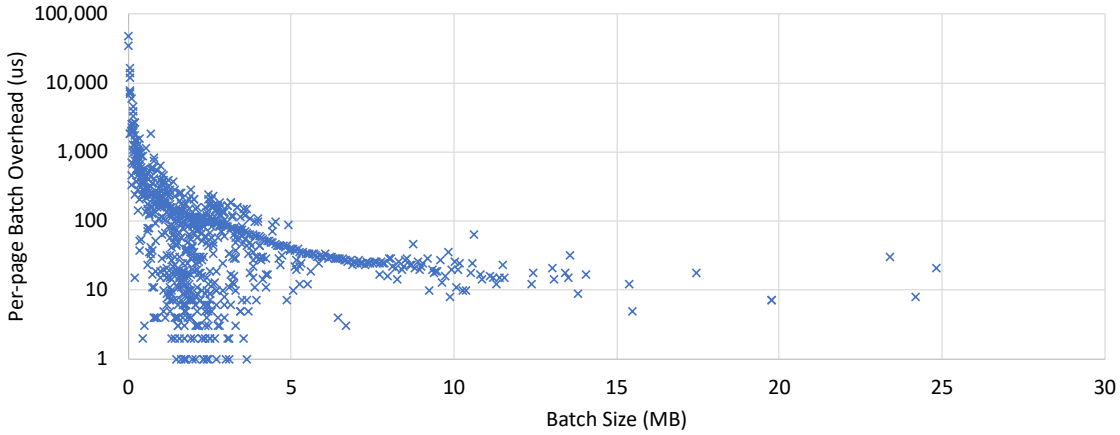


Figure 5.2: Per-page batch processing overhead (us) vs. batch size (MB).

Second, we observe that page evictions introduce unnecessary serialization in page migrations. We examine a GPU runtime implementation (NVIDIA driver v396.37 [16]) to understand how modern GPUs perform memory management and when the decision on eviction is made. When the GPU runtime handles a page fault, the physical memory allocator in the GPU runtime tries to allocate a new page (or a free root chunk) in the GPU memory (`alloc_root_chunk()`). If such an allocation fails, indicating that the GPU memory runs out of space, then a page eviction is requested (`pick_and_evict_root_chunk()`). The physical memory manager in the GPU runtime then checks whether it can evict any user memory chunks to satisfy the request[1]. Once a suitable root chunk is selected for eviction, its eviction flag is set (`chunk_start_eviction()`), and subsequently, the eviction begins

[1] All allocated user memory root chunks are tracked in an LRU list (`root_chunks.va_block_used`). A root chunk is moved to the tail of the LRU list whenever any of its sub-chunks is allocated. This is the policy referred to as aged-based LRU in literature [42, 43, 44]. To examine the head of the LRU list, `list_first_chunk()` is used.

(`evict_root_chunk()`). Once the eviction is completed, the metadata (e.g., tracker and status data) associated with the chunk is freed and the frame in the GPU memory becomes available. Subsequently, the new page migration begins.

From this, we conclude that a page eviction and a new page allocation are serialized in modern GPUs to prevent the new page from overwriting the evicted page. Note that an eviction is always required every time a page fault occurs once the GPU memory becomes full. Figure 5.3 depicts these operations. When the GPU runtime fails to allocate page A, it initiates an eviction of page X, reactively (❶). Once page X eviction from the GPU memory is completed, both the master page table in the CPU memory and the GPU page table are updated for the evicted page X and the frame is freed (❷). Once the frame is freed, page A migration begins (❸). In Section 5.3.2, we propose a technique to eliminate this serialization.

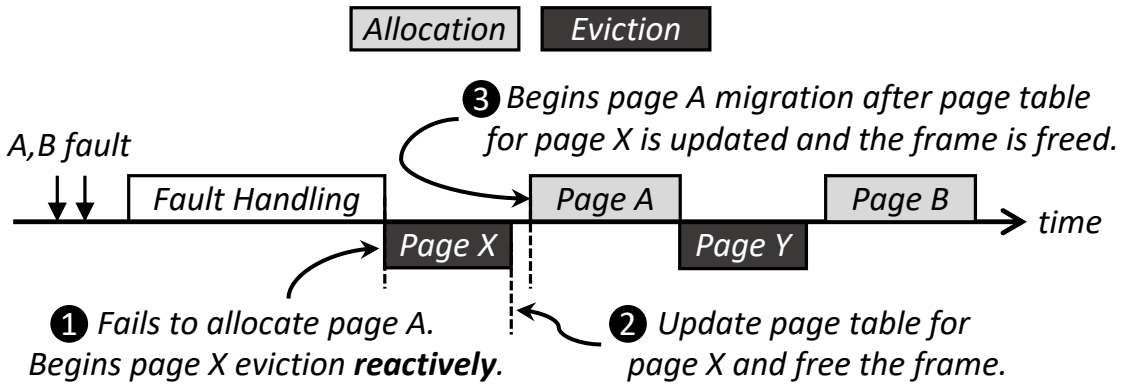


Figure 5.3: Overview of how and when GPU runtime evicts a page from GPU memory, and why it is on the critical path.

5.3 Challenges and Solutions

In this section, we present techniques to mitigate the inefficiencies described in Section 5.2. In Section 5.3.1, we propose a technique to increase the batch size in order to amortize the batch processing overhead and reduce the number of batches. In Section 5.3.2, we propose a technique to reduce the batch processing time by overlapping page evictions with

CPU-to-GPU page migrations, not relying on the timing prediction.

5.3.1 Thread Oversubscription

In GPUs, the primary execution unit is a warp, which is a collection of scalar threads (e.g., 32 in NVIDIA GPUs) that run in a single-instruction multiple-thread (SIMT) fashion. A warp is stalled once it generates a page fault. Provided that only up to 64 warps (or 2048 threads) can concurrently run in an SM [8, 9], it does not take much time before the GPU becomes crippled due to lack of runnable warps. The number of concurrently running threads has been engineered to provide enough TLP to hide memory latencies in traditional GPUs, where no page migrations between CPU and GPU occur. We find that the level of thread concurrency optimized for traditional GPUs is not sufficient to amortize the batch processing overhead in the presence of demand paging.

There are three approaches to increase the batch size. First approach is to increase memory-level parallelism (MLP) with out-of-order execution. Out-of-order execution has been used in CPUs and therefore studied extensively. However, adopting an out-of-order scheduler designed for CPUs to GPUs is not trivial. Foremost, CPUs and GPUs employ very different register file structures and access arbitration logic. GPU register files are statically partitioned among all of the threads scheduled to the same SM. To properly support the SIMT execution model, it is crucial to ensure that no bank conflicts occur. For this, GPU register files and the access arbitration logic are designed and optimized in a different way than that of CPUs. Therefore, it is not trivial to adopt a CPU-style out-of-order scheduler to a GPU. Second approach is to use the stalled warps to generate more page faults. For this, runahead execution [45] or speculative execution [46, 47, 48] techniques can be employed. However, these techniques are likely less effective to generate a large number of page faults in a short amount of time because each thread block typically runs short due to GPU programming model. Third approach is to increase thread concurrency by dispatching more thread blocks to an SM. However, the number of threads (or thread blocks) per SM is

dictated by the physical resource constraint. We want a solution that can increase thread concurrency without increasing physical resource requirement.

To this end, we develop thread oversubscription, a GPU virtualization technique. We utilize the Virtual Thread (VT) [49] as our baseline architecture for GPU virtualization. The VT architecture assigns thread blocks up to the capacity limit (i.e., physical resource constraints, such as register file and shared memory), while ignoring the scheduling limit (i.e., scheduler resource constraints, such as program counters and SIMT stacks). It dispatches thread blocks in an active and inactive states, such that the number of active thread blocks does not exceed the scheduling limit. Once all of the warps in an active thread block are descheduled due to long latency operations, the active thread block is context switched out, and the next ready but inactive thread block takes its place. Since both active and inactive thread blocks fit within the capacity limit, the need to save and restore large amounts of context (e.g., register files) is obviated.

Our primary objective is to increase the batch size to amortize the impact of batch processing rather than to increase the TLP. However, we find that vanilla VT is not applicable to most of our evaluated graph workloads as is because not enough resource is available to host even an additional thread block. The reason is that the number of thread blocks schedulable to an SM is often limited by the maximum number of threads per SM for most graph workloads. When the maximum number of threads per SM is scheduled to an SM, the maximum number of registers per SM is easily exhausted. Take NVIDIA Titan Xp [8] as an example. Provided that the maximum number of threads per SM is 2048 and the maximum number of registers per SM is 65536 [8], each thread can use up to 32 registers. If each thread uses more than 16 registers (i.e., a half of 32 registers), which is the case for most of our evaluated workloads, vanilla VT cannot host even a single additional thread block due to the register file resource constraint.

Hosting an additional (or more) thread blocks in an SM in this case requires more expensive context switching [49, 50, 51, 52]. This includes saving and restoring the per-

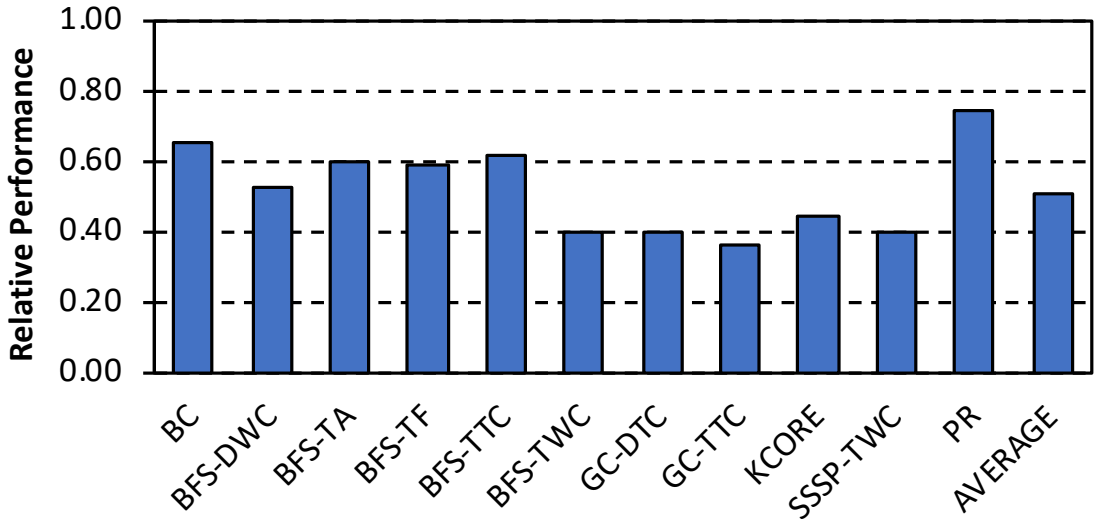


Figure 5.4: Performance overhead when provisioning an additional thread block to each SM requires context switching in traditional GPUs.

thread-block state information (e.g., warp identifiers, thread block identifiers, and SIMT stack including the program counter) and register files to the global memory. Note that using shared memory to store the context switching information is no longer feasible[2]. This cost is intolerable in traditional GPUs, where no page migrations between CPU and GPU memory occur. Figure 5.4 shows the performance impact assuming we provision an additional thread block to each SM with context switching in traditional GPUs. We see that the context switching overhead leads to a non-negligible performance degradation across all the evaluated workloads (49% on average). This indicates that the context switching overhead caused by adding an additional thread block to each SM outweighs the benefit of increasing thread concurrency if the running workloads fit in the GPU memory.

However, we observe that in the presence of page migrations between CPU and GPU memory, increasing thread concurrency is beneficial despite the expensive context switching overhead. To this end, we extend the VT in three ways. First, we employ an additional

[2]Assume each thread block consists of 2048 threads and each thread uses 10 32-bit registers. In this case, 85KB = 80KB (2048 * 10 * 4 bytes) for register files + 5KB for thread block state information has to be stored and restored for context switching. The size of the thread block state information is estimated according to [49]. According to [8, 9], shared memory size can be configured up to 64KB per SM. Therefore, it is infeasible to use shared memory for context switching.

mapping table so that different Virtual Warp IDs (VWIs) can access the same set of register files when they are context switched. Note that VWI is a unique warp identifier across all the assigned warps to an SM, including both active and inactive thread blocks [49]. Only when a thread block finishes execution, its VWIs are released and reused for another thread block. Second, we extend the operation performed by the Virtual Thread Controller (VTC), which keeps track of the state of all thread blocks in order to determine which thread block can be brought back to active from inactive state, or vice versa, when a thread block is swapped out. The vanilla VT only stores the per-thread-block state information in the shared memory through the context handler. We extend this operation to store register files as well. Given that the register files that a thread block uses can easily exceed tens of KBs, we use the global memory instead of the shared memory.

Third, we dynamically control the degree of thread oversubscription based on the rate at which premature eviction occurs. A premature eviction occurs when a page is evicted earlier than should be and a page fault is generated for the page again by the GPU. Since the thread oversubscription increases the number of concurrently running threads, it might lead to an increase in the working set size, which increases the likelihood of premature evictions. This is detrimental because when a premature eviction occurs, the evicted page has to be brought back to the GPU memory. On the other hand, an increase in thread concurrency can lead to better page utilization, reducing premature evictions. Hence, we modify the GPU runtime to monitor the premature eviction rates and dynamically control the degree of thread oversubscription. In Section 5.5.1, we provide a detailed analysis on the impact on premature eviction that the thread oversubscription causes.

Figure 5.5 shows how our thread oversubscription technique works. We enable thread oversubscription from the beginning of the execution by allocating one additional thread block to each SM (❶). The thread block additionally allocated to each SM is inactive at first. It is important to note that the number of active thread blocks does not exceed that of the baseline, which is determined by the physical resource constraints. Once all of the warps in

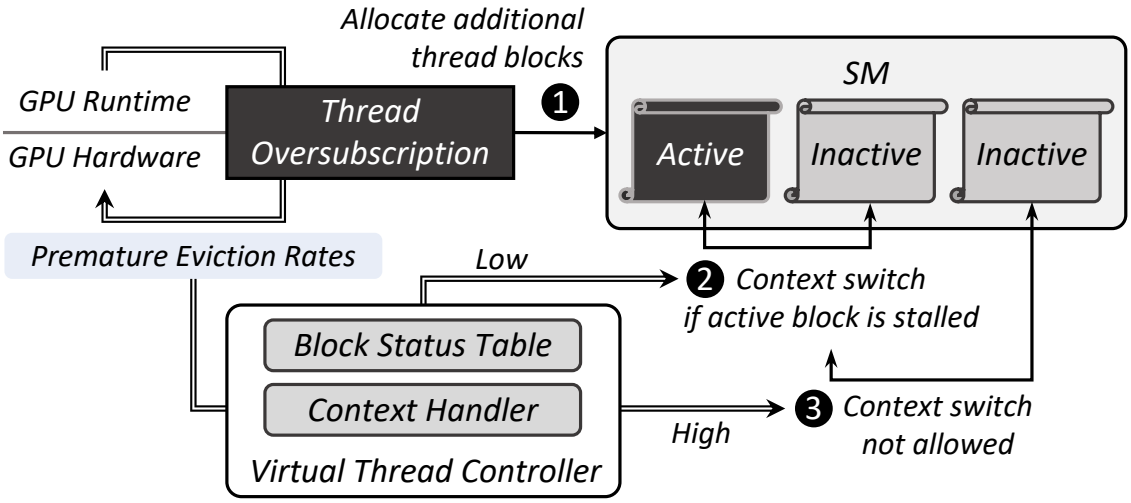


Figure 5.5: Thread oversubscription scheme.

an active thread block are stalled due to page fault, the thread oversubscription mechanism context switches the active (but stalled) thread block with an inactive thread block (❷). The thread oversubscription mechanism can be detrimental if it incurs premature evictions. To prevent this, the GPU runtime monitors the degree of premature eviction by periodically estimating the running average of the lifetime of pages by tracking when each page is allocated and evicted[3]. We use the running average as an indicator of premature evictions. If the running average is decreased by a certain threshold[4], the thread oversubscription mechanism does not allow any more context switching by decrementing (and limiting) the number of concurrently runnable thread blocks (❸)[5]. Otherwise, thread oversubscription allocates one additional thread block to each SM in an incremental manner.

Figure 5.6 demonstrates how our thread oversubscription technique can increase the batch size. For ease of explanation, we assume that up to one thread block (TB) can be dispatched to an SM. We also assume that page A, B, and C are accessed by TB1, and page D is accessed by TB2. In the baseline, TB2 can be executed only after TB1 is retired. With

[3]This is a tunable parameter. We calculate the running average of the lifetime of pages at every 100k cycles for our evaluation.

[4]The threshold is a tunable parameter and empirically set to 20% for our evaluation.

[5]The number of concurrently runnable thread blocks is set to the number of allocated thread blocks at first. This does not mean all of them are running simultaneously. The warp scheduler picks a warp from active thread blocks only.

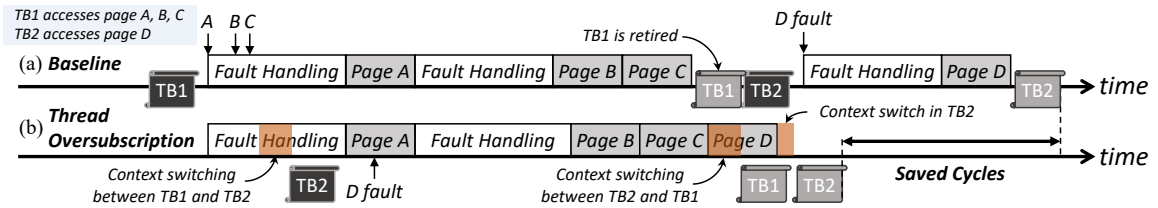


Figure 5.6: Overview of how thread oversubscription can increase the batch size. TB is short for thread block.

thread oversubscription, TB1 is context switched with TB2 when all of its warps are stalled (after page fault for page C is generated). After context switching overhead, TB2 is executed and a page fault for page D is generated, which can be handled along with those for page B and C. Once page C migration is done, TB1 can be resumed. After context switching overhead, TB1 is resumed and retired. Once page D migration is done, TB2 is context switched in, resumed, and retired. As can be seen in the figure, the thread oversubscription eliminates the need for the third batch processing, reducing the overall batch processing overhead. Note that the batch processing overhead for the second batch is slightly increased to handle an additional page fault for page D.

5.3.2 Unobtrusive Eviction

While the thread oversubscription technique alleviates the performance impact of demand paging by reducing the number of batches and amortizing the batch processing overhead, there is an opportunity to reduce the batch processing time itself. Figure 5.7 shows the performance impact of the page eviction by comparing the performance of a GPU with 50% memory oversubscription (denoted as baseline) to the performance of a GPU with unlimited memory, where no page evictions occur. The GPU with 50% memory oversubscription experiences an average performance loss of 46%. We compare this to the performance of a GPU with an instant page eviction capability (denoted as ideal eviction). Removing the page eviction latency achieves an average performance improvement of 16%. To this end, we propose unobtrusive eviction, a mechanism that eliminates the page eviction latency by

overlapping page evictions with page migrations to the GPU.

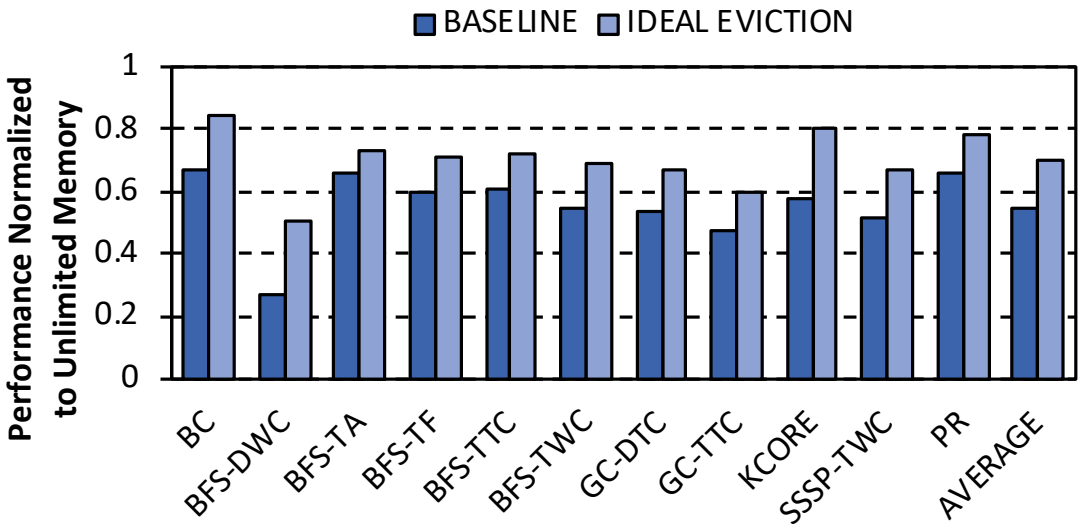


Figure 5.7: Performance of a GPU with 50% memory oversubscription compared to a GPU with unlimited memory.

We exploit the bidirectional transfers allowed in the DMA engines used in modern CPUs and GPUs [41, 53, 54, 55, 56]. However, a page eviction and a new page allocation are serialized in modern GPUs to prevent the new page from overwriting the evicted page, despite the available bidirectional transfers in the DMA engines, as discussed in Section 5.2. Our goal is to devise a mechanism that exploits the bidirectional transfers without violating the serialization requirement. The key idea is to preemptively initiate a single page eviction. To perform this preemptive eviction promptly at the beginning of the batch processing, we modify the GPU runtime and add a new GPU memory status tracker. This does not affect page fault handling performance since the tracking is performed by the GPU runtime only when a new page is allocated in the GPU memory.

When a page fault interrupt is raised by the GPU MMU, the top-half interrupt service routine (ISR) responds. It checks if the GPU memory runs out of space via the GPU memory status tracker. If so, then it sends a preemptive eviction request to the GPU. The rest of the batch processing (e.g., preprocessing of the page faults, CPU-side page table walks) is performed by the bottom-half ISR. Provided that the batch processing overhead (i.e., the

time between when a batch processing begins and when the first page migration of the batch begins) is at least tens of microseconds, the single page eviction is completed before the first page migration begins. By doing so, the first page migration can proceed without any delay. If a subsequent page eviction is required, the bottom-half ISR of the GPU runtime schedules the next page eviction along with the page migration to the GPU memory. Note that since the single page preemptive eviction is initiated by the GPU runtime when the batch processing begins, no additional overhead (e.g., communications with GPU) is required. Figure 5.8 shows how the unobtrusive eviction is implemented.

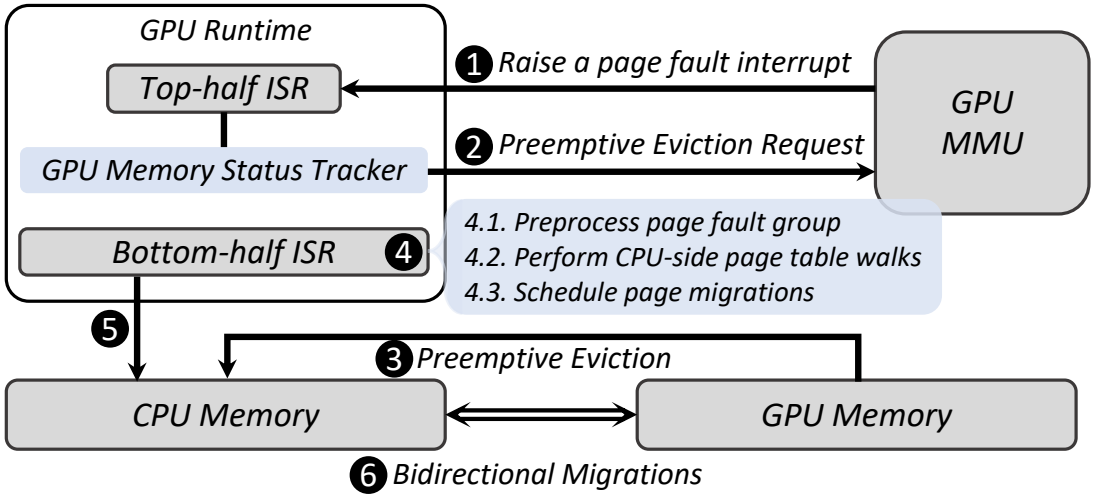


Figure 5.8: Unobtrusive eviction scheme.

Figure 5.9 provides a timeline example of how the unobtrusive eviction works. When the GPU runtime begins a batch processing, it checks the GPU memory status. If it runs out of space, it initiates a single page eviction (❶). Once page X eviction from the GPU memory is completed, both CPU and GPU page tables are updated (❷). Unlike in the case of the baseline (Figure 5.3), page A can be migrated to the GPU memory without any delay (❸). At the same time, page Y can be evicted using the bidirectional transfers. Since the data transfer speed from GPU to CPU memory is faster than the other way around [42], eviction is completely unobtrusive and migrations to the GPU can occur without any delay.

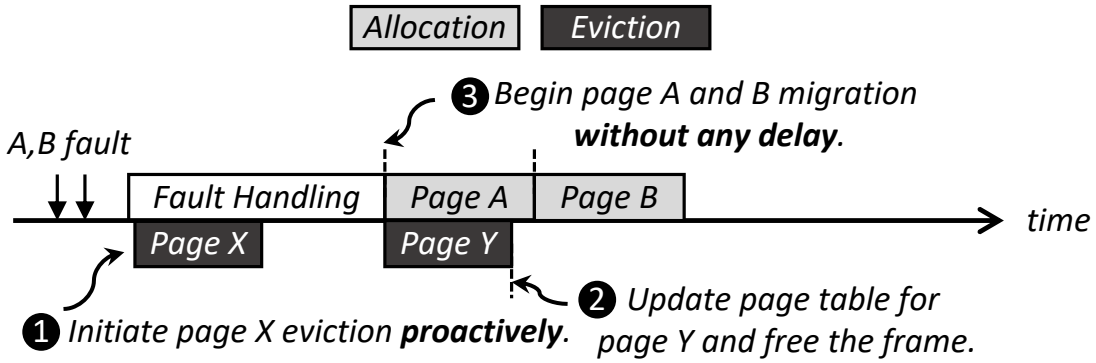


Figure 5.9: Overview of how unobtrusive eviction works.

5.4 Evaluation

In this section, we describe our evaluation methodology and evaluate our proposed techniques. We provide more detailed analyses in the following section (Section 5.5).

5.4.1 Methodology

Simulator. We use MacSim [25], a cycle-level microarchitecture simulator. We modify the simulator to support virtual memory, demand paging [8, 9, 56, 43, 44, 29, 13, 14, 42], and the Virtual Thread (VT) [49]. Our virtual memory implementation includes TLBs, page tables, and a highly-threaded page table walker [11, 12, 29]. Demand paging is modeled based on the GPU runtime software for NVIDIA PASCAL GPUs (driver v396.37) [16, 55, 8, 56, 43, 44]. We also model the batch processing mechanism that remedies a plethora of fault buffer entries together. We use a 1024-entry fault buffer [56] to handle up to a thousand of simultaneous page faults. The page table walker is shared across all the SMs in the GPU, allowing up to 64 concurrent page walks [12]. Each TLB contains the miss-status-holding-registers (MSHRs) to track in-flight page table walks [29].

To be conservative for the benefit of our mechanism, we use a lower-bound 20μs for the batch processing overhead among the ones used in other works [41, 29, 13, 14, 42]. We also employ the state-of-the-art page prefetching mechanism [29]. The GPU memory capacity is

configured to be fractions (50% and 75%) of the memory footprint of each workload as in prior works [29, 42]. We use LRU for page replacement policy [56, 43, 44]. We include the context switching overhead (i.e., timing overhead of storing and restoring context, such as register files and per-thread-block state information, to/from global memory whenever a context switching occurs) in the evaluation for thread oversubscription.

We also evaluate the unobtrusive eviction using simulator, in which we faithfully model the scheme described in Section 5.3.2. We were not able to implement the mechanism in real NVIDIA runtime software because the open source part of the runtime software does not include the part where it interacts with the driver entirely.

We measure the footprint of each application and set the GPU memory size such that 50% of the application footprint can fit in the GPU memory. We perform a sensitivity study on the oversubscription rates by changing the memory size.

Table 5.1: Configuration of the simulated system

GPU Configuration	
Core	16 SMs, 1024 threads per SM, 256KB register files per SM
Private L1 Cache	16KB, 4-way, LRU, L1 misses are coalesced before accessing L2
Private L1 TLB	64 entries per core, fully associative, LRU

Memory Configuration	
Shared L2 Cache	2MB total, 16-way, LRU
Shared L2 TLB	1024 entries total, 32-way associative, LRU
Memory	200 cycle latency

Unified Memory Configuration	
Fault Buffer	1024 entries
Fault Handling	64KB page size, 20μs batch processing overhead, 15.75GB/s PCIe bandwidth

Workloads. We select 11 workloads from the GraphBIG benchmark suite [20]. These include Betweenness Centrality (BC), Breadth-First Search (BFS), Graph Coloring (GC), Single-Source Shortest Path (SSSP), K-core decomposition (KCORE), and Page Rank (PR). BC is an algorithm that detects the amount of influence a node has over the flow of information in a graph [57]. Graph traversal is the most fundamental operation of graph computing, for which we include five different implementations of BFS: data-warp-centric (DWC), topological-atomic (TA), topological-frontier (TF), topological-thread-centric (TTC), and topological-warp-centric (TWC). GC performs the assignment of labels or colors to the graph elements (i.e., vertices or edges) subject to certain constraints [58, 59], for which we include two different implementations: data-thread-centric (DTC) and topological-thread-centric (TTC). KCORE partitions a graph into layers from external to more central vertices [60]. SSSP finds the shortest path from the given source to each vertex, for which we include a topological-warp-centric (TWC) implementation. PR is an algorithm that evaluates the importance of web pages [61]. The footprints of these workloads range from 26MB to 349MB with an average of 74MB. Impractically long simulation times prevent us from using the entire real-world data set. Note, however, that the footprints are larger than the ones used in prior works [29, 42].

5.4.2 Performance

Figure 5.10 shows the performance of the proposed thread oversubscription and unobtrusive eviction mechanisms (denoted as TO and UE, respectively). The performance is normalized to that of a baseline (denoted as BASELINE) that uses the state-of-the-art page prefetching technique, proposed by Zheng et al. [29]. We also evaluate how the performance changes when PCIe (de)compression is utilized (denoted as BASELINE with PCIe Compression). Lastly, we compare our mechanisms with the eviction-throttling-compression (ETC) mechanism [42]. Our mechanism (TO+UE) achieves an average speedup of 2x and 1.81x relative to the BASELINE and BASELINE with PCIe Compression, respectively. Our mechanism

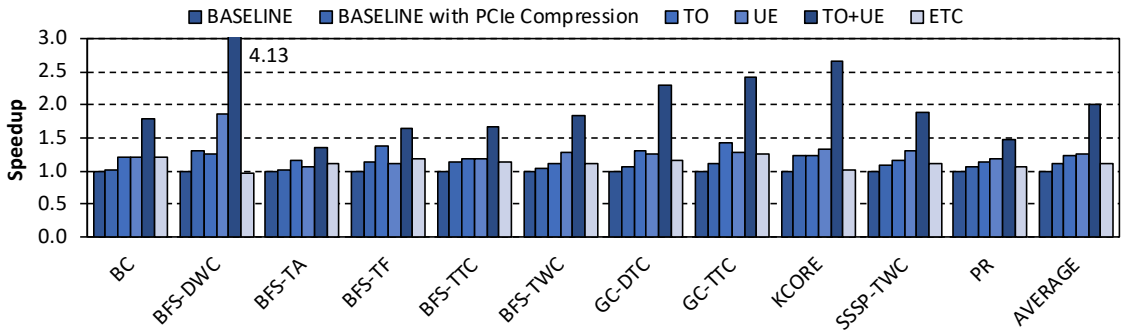


Figure 5.10: Performance comparison among baseline with the state-of-the-art page prefetching [29], eviction-throttling-compression (ETC) [42], and our proposed mechanisms (thread oversubscription is denoted as TO, and unobtrusive eviction is denoted as UE), normalized to the baseline.

even outperforms ETC by 79% on average. ETC includes three components: proactive eviction (PE), memory-aware throttling (MT), and capacity compression (CC). For irregular applications, however, PE is disabled because the proposed PE hurts performance[6]. The memory-aware throttling (MT) technique can be beneficial when memory is oversubscribed if it can reduce the working set size and thus decrease the page thrashing rates[7]. However, as we discussed in Section 5.1, since most of the memory pages are shared in many large-scale, irregular workloads, MT is not effective.

The performance improvement achieved by our mechanism is mainly attributed to the following two key factors. First, as described in Section 5.3.1, our thread oversubscription technique effectively reduces the total number of batches, thereby mitigating the overall fault handling overhead. For the evaluated workloads, we see that the number of batches is reduced by 51% on average because we process a 2.27x more number of page faults in per batch, as shown in Figures 5.11 and 5.12. This corroborates our observation that it is beneficial to increase thread concurrency in the presence of frequent page migrations, even at the cost of expensive context switching.

Second, the unobtrusive eviction technique reduces the average batch processing time

[6]We faithfully model ETC to the best of our knowledge.

[7]When triggered, MT statically throttles half of the SMs in the beginning. After the initial phase, it repeats two epochs, the detection epoch and the execution epoch. Depending on the behavior monitored during the detection epoch, MT decides to throttle or unthrottle SMs.

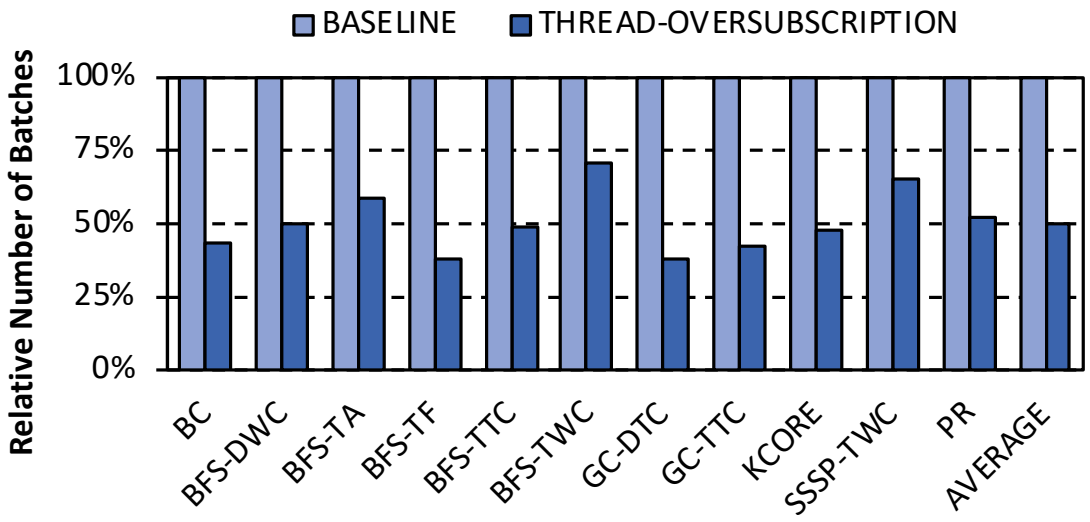


Figure 5.11: Total number of batches.

significantly. As seen in Figure 5.13, when employed with the thread oversubscription technique, it reduces the average batch processing time by 60%. The unobtrusive eviction technique is particularly effective for `BFS-DWC`. From further investigation, we see that `BFS-DWC` has an extremely high divergent memory access pattern. Because of this, constant page thrashing occurs throughout the execution. Therefore, the unobtrusive eviction, which hides the eviction latency, leads to a 4.24x performance improvement.

When both of them are employed (denoted as TO+UE), we find that the average batch processing time is reduced by 27% compared to the baseline even though we handle more page faults per batch. As results, the thread oversubscription technique achieves 22% performance improvement and the unobtrusive eviction technique achieves 61% additional performance improvement, as shown in Figure 5.10.

5.5 Analysis

In this section, we provide in-depth analyses on our proposed mechanisms.

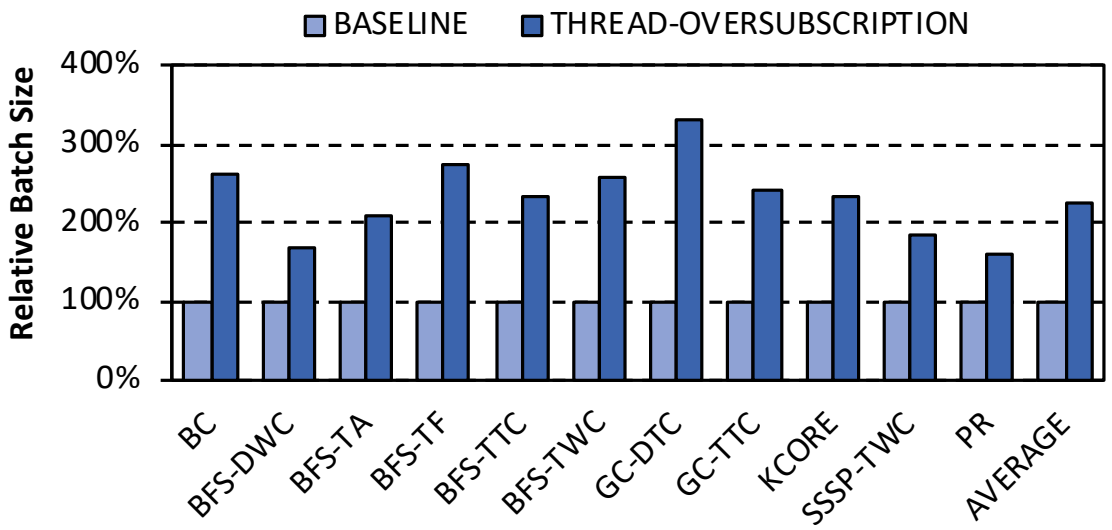


Figure 5.12: Average batch sizes.

5.5.1 Effect on Premature Eviction

A premature eviction occurs because memory capacity is smaller than working set. Since the thread oversubscription technique increases the batch size, premature evictions can also increase. Therefore, premature eviction is a key metric to evaluate the efficacy of the thread oversubscription technique. Figure 5.14 shows the premature evictions when the thread oversubscription technique is employed, compared to the baseline. Surprisingly, the thread oversubscription technique decreases premature evictions in most of the workloads. The reason is that it increases the likelihood of GPU resident pages being used before evicted. This is particularly the case for topological graph workloads since the input graph is traversed or processed in a topological order by each thread block in the topological graph workloads. Since the thread oversubscription technique increases thread concurrency and makes their memory accesses more parallel, there is a higher chance of concurrently running thread blocks accessing similar sets of pages while they are resident in the GPU memory.

The only exception is `BFS-TWC`. This happens when the working sets of existing thread blocks and additional thread blocks (context switched in) are distinct (as in the case of `BFS-TWC`), thrashing each other. We observe that the premature eviction is increased as

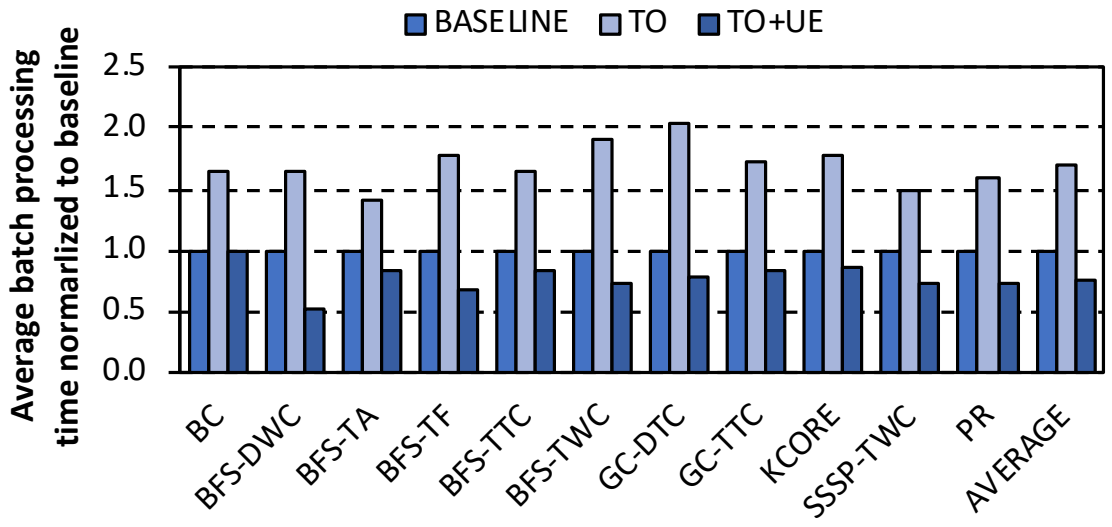


Figure 5.13: Average batch processing time.

we increase the number of concurrently runnable thread blocks for some workloads (e.g., `BFS-TWC` and `SSSP-TWC`). In the case of `BFS-TWC`, the premature eviction is increased by 4% and 38% compared to the baseline as we increase the number of concurrently runnable thread blocks by 2x and 3x, respectively. In the case of `SSSP-TWC`, the premature eviction is decreased by 2% but increased by 27% compared to the baseline as we increase the number of concurrently runnable thread blocks by 2x and 3x, respectively. As shown in Figure 5.14, however, this detrimental effect is delimited since our technique monitors the premature eviction rates and dynamically controls the degree of thread oversubscription, as described in Section 5.3.1.

5.5.2 Effect on Batch Size

Figure 5.15 compares the distribution of batch size. Batch size is computed as the summation of all the pages in each batch in size. Efficiency is computed as the reciprocal of an average time to handle each page. The line chart shows that as the number of page faults per batch is increased, efficiency is increased since the batch processing overhead is amortized. It is clearly seen that bigger batches appear when the thread oversubscription is employed. From

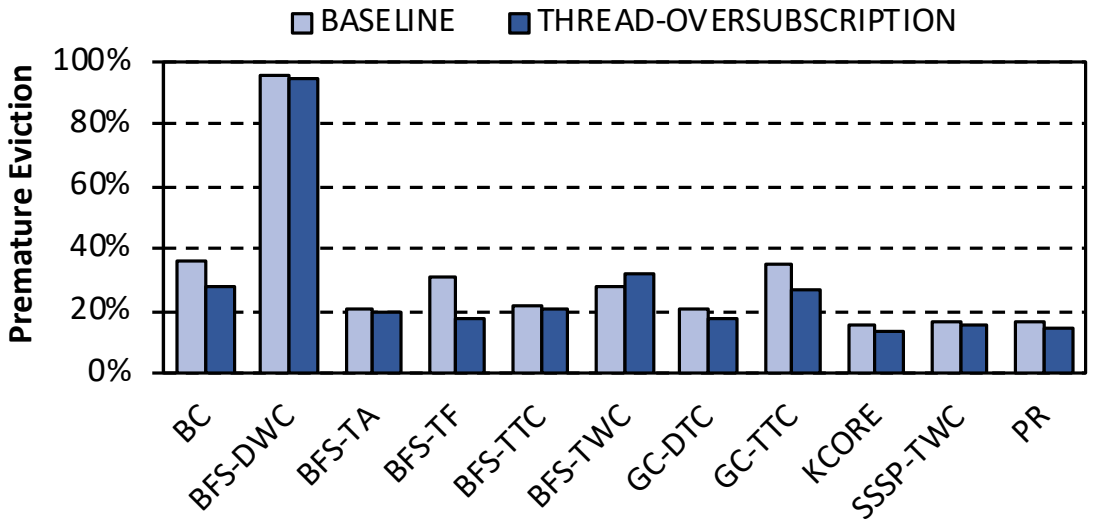


Figure 5.14: Premature eviction comparison.

this, we conclude that thread oversubscription effectively increases the batch size.

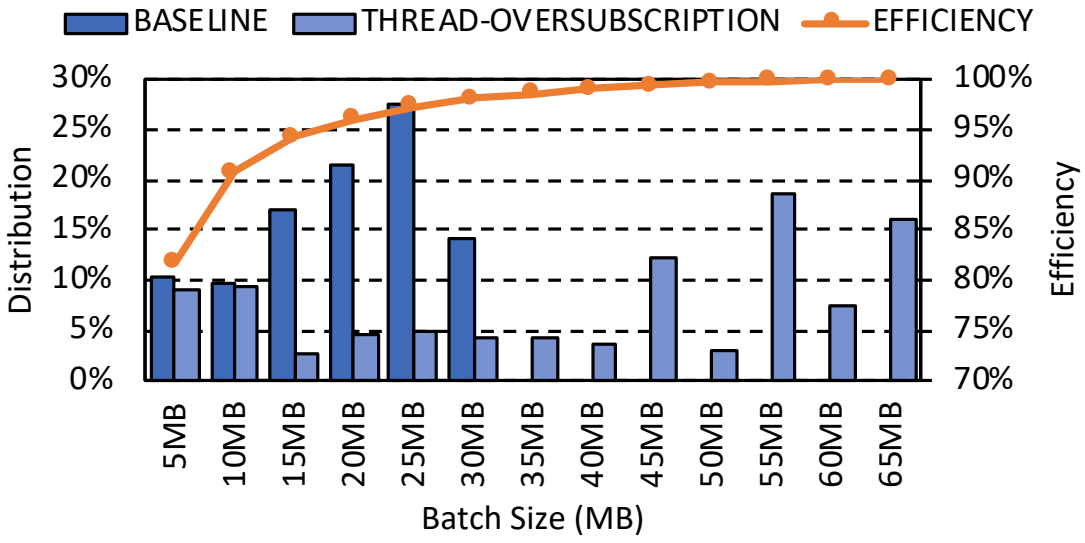


Figure 5.15: Batch size comparison.

5.5.3 Sensitivity to Oversubscription Ratio

Figure 5.16 shows the performance impact of memory oversubscription (bar chart) and the performance improvement of the unobtrusive eviction technique (line chart) when memory

oversubscription ratio is varied from 0.1 (i.e., the GPU physical memory capacity is set to 10% of each application's working set size) to 1.0 (i.e., all application data fits in the GPU memory). We observe the followings. First, the performance impact of memory oversubscription increases as the GPU memory becomes smaller. With smaller GPU memory, smaller fraction of application fits in the GPU memory. This causes more frequent page evictions to occur, requiring more page migrations between CPU and GPU. Second, the unobtrusive eviction technique provides a scalable performance benefit. Obviously, when all applications data fits in the GPU memory, it is ineffective (i.e., speedup of 1). As the memory size becomes smaller, its efficacy increases, as page evictions occur more frequently. It provides an average speedup of 1.63x when oversubscription ratio is 0.1. Therefore, we conclude that the unobtrusive eviction technique can provide commensurate amount of performance benefit as the application requires more memory.

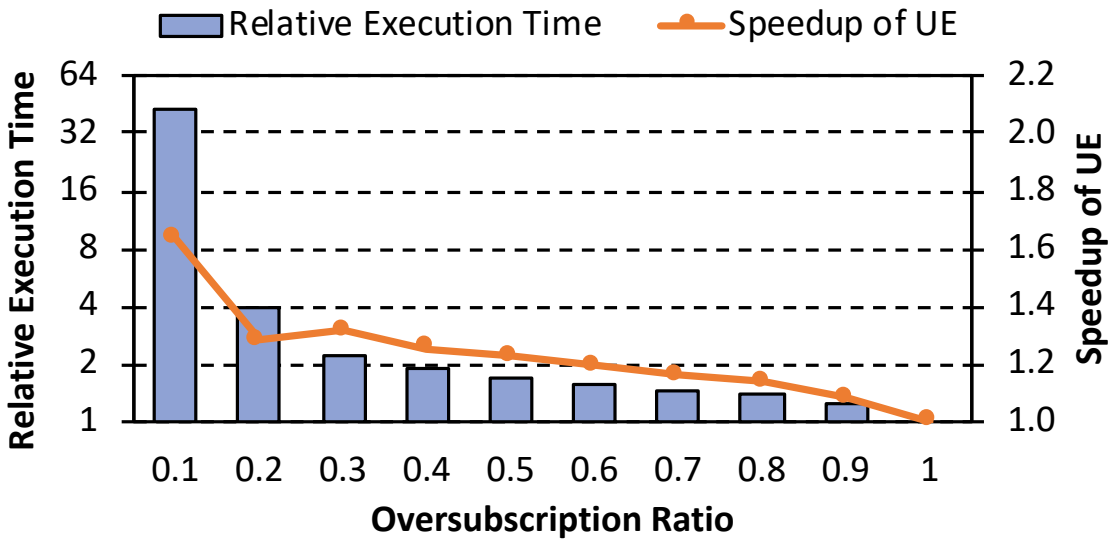


Figure 5.16: Sensitivity to memory oversubscription ratio.

5.5.4 Sensitivity to Batch Processing Overhead

Figure 5.17 shows average performance improvement of all workloads when batch processing overhead is varied from 20μs to 50μs, normalized to when the batch processing

overhead is 20μs. The data shows that the performance improvement that our proposed techniques can bring increases as the batch processing overhead becomes larger. To give perspective, we profile a regular application (e.g., `vectoradd`) as well in addition to the BFS measurement (Section 5.3.1). We observe that the batch processing overhead is higher for an irregular application (with a minimum of 50μs) than for a regular application (with a minimum of 20μs). Provided that a batch processing usually takes more than 300μs in real GPUs, we believe that our proposed techniques can bring significantly high performance improvement when employed in real GPUs.

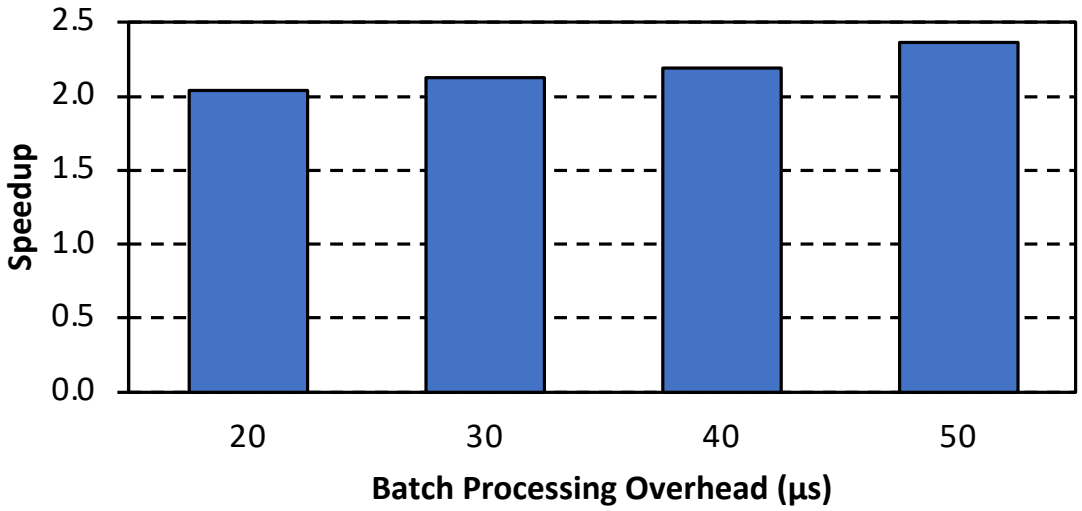


Figure 5.17: Sensitivity to batch processing overhead.

5.5.5 Context Switching Overhead

For TO evaluation, we include the context switching overhead (i.e., timing overhead of storing and restoring context, such as register files and per-thread-block state information, to/from global memory whenever a context switching occurs). Although not shown, we also evaluated a close-to-ideal context switching overhead using an infinite-size shared memory. For this, we used latencies computed according to [62]. From this, we observed that the overall execution time is insensitive to the context switching overhead.

CHAPTER 6
RELATED WORK

6.1 Multi-GPU Systems

Static-time data allocation has been researched in the context of multiple GPUs. A system with multiple GPUs is similar to an Message Passing Interface (MPI) based system since each GPU has its own memory and physical address space is not interleaved across multiple GPU memories. In this sense, several algorithms were proposed to automatically partition data among multiple devices (e.g., multiple GPUs, or CPUs and GPUs) [63, 64, 65, 66, 67, 68, 69]. The focus of our work (Section 3) is to enable data partitioning among GPU memories via selective use of coarse-grain interleaving (hardware mechanism) and to enable co-location of compute with the data they access (software mechanism). Ziabari et al. [70] have proposed a mechanism that supports seamless data transfer across all the devices (a CPU and one or more GPUs) in the system, while creating a hierarchical view between the memory of the GPUs and the host memory. Kim et al. [71] have proposed a GPU memory network to simplify data sharing between discrete GPUs.

Arunkumar et al. [4] have demonstrated that package-level integration of multiple GPU modules (GPMs) can enable continuous performance scaling and proposed a technique to improve GPM data locality and minimize the sensitivity on inter-GPM bandwidth. They used the PASCAL and later architectures where unified memory and demand paging are supported, in which when a page is first accessed in a kernel, a page fault is detected and the page fault handling procedure is performed. Realizing the first-touch based allocation on those architectures is not revolutionary since it can be done by modifying GPU driver where page allocation is performed. However, it requires applications to use unified memory in the first place, which indicates that traditional GPU applications, in which CPU allocates data

and GPU consumes or processes it after kernel(s) are launched, cannot gain the benefit of the mechanism as is. On the other hand, our mechanism does not require any application modifications, supporting any applications for better compute and data co-placement.

6.2 Memory-Level Parallelism

Zurawski et al. [72] have presented an address bit swapping scheme to increase memory-level parallelism by reducing the row buffer conflicts in traditional DRAM systems, which is used in AlphaStation 600 5-series workstations. Zhang et al. [18] have proposed a permutation-based page interleaving scheme in order to reduce row-buffer conflicts and to exploit data access locality in the row-buffer. Ghasempour et al. [19] have proposed a hardware mechanism to dynamically change the address mapping to increase bank-level parallelism at the cost of a significant amount of page migration overhead. While our proposed mechanism (Section 3) also uses address bit swapping scheme, it is different from these works in two ways. First, our mechanism applies address mapping scheme at a page granularity such that pages with different address mappings co-exist in the same memory space. Our mechanism is lightweight in a sense that it incurs negligible performance overhead and does not have any impact on the cache coherence protocol or virtual address translation. Second, our mechanism does not require large-scale page migrations; only a few (e.g., four or eight, depending on the number of memory stacks) pages are affected, since we selectively use coarse-grain interleaved page (CGP) at the page-group granularity.

6.3 Static-time Data Alignment

Static-time data allocation has a long history of research. HPF (High Performance Fortran) provides compiler directives to specify data alignment among processors [73]. Although our mechanism shares the same philosophy with the HPF directives such as block or cyclic, they are different in the sense that the HPF directives are applied at virtual address space, whereas it is done in the physical memory space in our mechanism since the source of non-

uniformity of memory access pattern is caused when a virtual page is mapped to the physical memory domain. Sung et al. [74] have presented a formulation and language extension that enables automatic data layout transformation for structured grid codes in Compute Unified Device Architecture (CUDA). It distributes concurrent memory requests evenly to DRAM channels and banks, thereby achieving significant speedup. Thanh-Hoang et al. [75] have recently proposed an architectural solution called Data Layout Transformation (DLT) for optimizing data movement across system components. While their accelerator can make good use of memory bandwidth for data movement, it requires application changes to use their instructions. Our mechanism, on the other hand, does not require any application modifications and provides high data locality with slight changes in virtual to physical address mapping.

6.4 Processing in Memory

Processing in memory (PIM) was proposed decades ago [76, 77, 78, 79, 80, 81, 82, 83, 84]. Recent advances in 3D stacking technology have given a boost to PIM research [85, 86, 87, 7, 6, 88, 5, 89, 90, 91, 92, 93, 94] to accelerate workloads in various domains (e.g., large-scale graph processing workloads [86, 89], Map-Reduce workloads [95], and HPC applications [6]). Akin et al. [87] have proposed solutions for efficient data reorganization, combining a DRAM-aware reshape accelerator integrated within 3D-stacked DRAM, and a mathematical framework that is used to represent and optimize the reorganization operations. Hsieh et al. [5] (TOM) have addressed the issue of local and remote memory accesses in a system with multiple PIM memory stacks. It performs runtime profiling to learn best address mapping for data accessed by offloading candidates and distributes that data with the discovered mapping. Although our work focus is on a system with multiple GPUs, it is worth comparing our mechanism against theirs in the context of a system with distributed GPUs, irrespective of whether they are multiple full-fledged GPUs or GPUs in memory stacks. In contrast to our proposal, this work 1) essentially delays and decelerates the regular

kernel execution because it tests all different address mappings (10 mappings, sweeping from bit position 7 to bit position 16) for all the data accessed by offloading candidates during the runtime learning phase, 2) implicitly assumes a hardware mechanism to distribute data with different mappings.

6.5 Virtual Memory Support in GPUs

Address translation is required for memory references in virtualized memory. The performance implications of address translation are widely known and considerable research has been done to reduce the overheads [96, 15, 97, 98, 99, 100, 101, 102, 103, 11, 12, 104, 62]. Power et al. [12] have studied memory management unit design for GPUs and proposed a TLB per compute unit and a shared highly threaded page table walker and page walk cache among compute units. Pichai et al. [11] have also explored GPU memory management units and proposed modest TLB and page table walker augmentations. Ausavarungnirun et al. [13] have proposed a GPU memory manager that provides an application-transparent multiple page size support in GPUs to increase the TLB reach. Ausavarungnirun et al. [14] have proposed an address-translation-aware GPU memory hierarchy design that reduces the overhead of address translation by prioritizing memory accesses for page table walks over data accesses. Shin et al. [105] have proposed a SIMT-aware page table walks scheduling mechanism to improve address translation performance in irregular GPU workloads. Cong et al. [106] have proposed a two-level TLB design for a unified virtual address space between the host CPU and customized accelerators.

6.6 Demand Paging in GPUs

Unlike traditional GPUs, where GPU runtime must migrate all memory pages to the GPU memory before launching kernel, modern GPUs support on-demand page faulting and migration (i.e., demand paging) [8, 9, 56, 43, 44]. The demand paging support eliminates the need for manual data migration, reducing programmer effort and enabling GPU applications

to compute across datasets that exceed the GPU memory capacity. However, its implication on performance is considerable and has been studied a lot recently [107, 108, 109, 110, 29, 13, 14, 42]. Zheng et al. [29] have explored the problem of PCIe bus being underutilized for demand-based page migration and proposed a software page prefetcher to better utilize PCIe bus bandwidth and hide page migration overheads. Agarwal et al. [108] have investigated the problem of demand (i.e., page access frequency) based page migration policy and proposed to use virtual-address-based program locality to enable aggressive prefetching and achieve bandwidth balancing. Agarwal et al. [109] have further explored page placement policies and proposed to use characteristics (i.e., bandwidth) of heterogeneous memory systems and program-annotated hints to maximize GPU throughput on a heterogeneous memory system.

Li et al. [42] have proposed ETC, a memory management framework to improve GPU performance under memory oversubscription. The ETC framework categorizes applications into three categories (regular applications with and without data sharing, and irregular applications) and apply three techniques (proactive eviction, memory-aware throttling, and capacity compression) differently. Their memory-aware throttling and capacity compression techniques are orthogonal to our work.

CHAPTER 7

CONCLUSION

Off-chip data migration has a significant impact on performance in modern GPU systems that employ multi-GPUs and/or utilize system memory. While the use of multi-GPUs in a system allows scalable compute capability, it is limited by techniques developed for traditional single GPU systems. The use of system memory provides an order of magnitude larger memory capacity to a GPU application, but it incurs intolerable performance degradation when used without care. This dissertation presents several ideas to help mitigating the data migration overhead.

In multi-GPU systems, it is crucial to co-locate compute and data together in a single GPU to exploit its large local memory bandwidth. Chapter 3 makes an observation that two key techniques developed in traditional GPU systems, fine-grain memory interleaving and thread block scheduling, are at odds with this. To enable co-placement of compute and data in the presence of fine-grain interleaved memory with a low-cost approach, we propose a mechanism that identifies exclusively accessed data and place the data along with the thread block that accesses it in the same GPU. The key ideas are (1) the amount of data exclusively used by a thread block can be estimated, and that exclusive data (of any size) can be localized to one GPU with coarse-grain interleaved pages, (2) using the affinity-based thread block scheduling policy, compute and data can be co-placed together, and (3) by using dual address mode with lightweight changes to virtual to physical page mappings, different interleaved memory pages can be selectively chosen for each data structure. The proposed mechanism improves performance by 31% and reduces 38% remote traffic over a baseline system.

While virtual memory support substantially reduces programmer's burden on running large-scale GPU applications, it has a significant performance implication. Chapter 5 finds

that (1) the page fault handling mechanism employed in contemporary GPUs introduces a large scale serialization in page fault handling, greatly hurting GPU throughput, and (2) page migrations between CPU and GPU are serialized and account for another significant fraction of page fault handling. To alleviate these inefficiencies and enable efficient demand paging for GPUs, we propose a GPU runtime software and hardware holistic solution that (1) reduces the number of batches (i.e., a group of page faults handled together) and amortizes the batch processing overhead by supporting CPU-like thread block context switching, (2) takes page eviction off the critical path with no hardware changes by overlapping evictions with CPU-to-GPU page migrations. The proposed solution provides an average speedup of 2.03x over the state-of-the-art page prefetching. It reduces the total number of batch processing by 51% on average and amortizes the batch processing overhead by increasing the batch size by 2.27x. The average batch processing time is reduced by 27%.

REFERENCES

[1] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[3] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[4] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[5] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[6] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014.

[7] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, "A New Perspective on Processing-in-memory Architecture Design," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2013.

[8] NVIDIA Corp., *NVIDIA Tesla P100*, `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016.

[9] ——, *NVIDIA Tesla V100*, `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[10] Advanced Micro Devices Inc., *AMD Graphics Cores Next (GCN) Architecture*, https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012.

[11] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[12] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[13] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[14] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[15] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.

[16] NVIDIA Corp., *NVIDIA Driver Downloads*, https://www.nvidia.com, 2018.

[17] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric System Interconnect Design with Hybrid Memory Cubes," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[18] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2000.

[19] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "DReAM: Dynamic Rearrangement of Address Mapping to Improve the Performance of DRAMs," in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2016.

[20] L. Nai, Y. Xia, I. G. Tanase, K. Hyesoon, and C.-Y. Lin, "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," in *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2015.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *IMPACT Technical Report*, 2012.

[23] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[24] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. A. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "The Structural Simulation Toolkit," *ACM SIGMETRICS Performance Evaluation Review*, 2011.

[25] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, T. Pho, H. Kim, and R. Hadidi, *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*, 2012.

[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 10, no. 1, pp. 16–19, 2011.

[27] *High bandwidth memory (hbm)*, JESD235A, 2015.

[28] NVIDIA Corp., *NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*, `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[29] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards High Performance Paged Memory for GPUs," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[30] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *Proceedings of the USENIX Conference on Security Symposium (SEC)*, 2016.

[31] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2011.

[32] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers," in *Proceedings of the*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1994.

[33] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[34] Cozlink, *The Differences between PCI Express 1.0, 2.0 And 3.0*, https://www.cozlink.com/pice-a272-2387-2388/article-73599.html, 2013.

[35] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, 1966.

[36] L. A. Belady and F. P. Palermo, "On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm," *IBM Journal of Research and Development*, 1974.

[37] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[38] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

[39] IBM, *GPUs Cloud Computing*, https://www.ibm.com/cloud/gpu.

[40] Google, *Google GPUs Cloud Computing*, https://cloud.google.com/gpu.

[41] PCI-SIG, *PCI Express Base Specification Revision 3.1a*, 2015.

[42] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A Framework for Memory Oversubscription Management in Graphics Processing Units," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[43] N. Akhenykh, *Unified Memory On Pascal and Volta*, http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf, 2018.

[44] ——, *Everything You Need to Know About Unified Memory*, http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[45] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

[46] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, "Speculative Execution on GPU: An Exploratory Study," in *International Conference on Parallel Processing (ICPP)*, 2010.

[47] X. Fan, S. Li, and W. Zhiying, "HVD-TLS: A Novel Framework of Thread Level Speculation," in *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012.

[48] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A Survey on Thread-Level Speculation Techniques," *ACM Comput. Surv.*, vol. 49, no. 2, 22:1–22:39, Jun. 2016.

[49] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[50] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[51] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[52] Z. Lin, L. Nyland, and H. Zhou, "Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[53] AMD, *AMD Accelerated Processing Units*, `https://www.amd.com/us/products/technologies/apu/Pages/apu.aspx`, 2011.

[54] ——, *AMD Graphics Cores Next (GCN) Architecture*, `https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, 2012.

[55] NVIDIA Corp., *CUDA Toolkit Documentation*, `https://docs.nvidia.com/cuda/index.html`, 2017.

[56] Peng Wang, *UNIFIED MEMORY ON P100*, `https://www.olcf.ornl.gov/wp-content/uploads/2018/02/SummitDev_Unified-Memory.pdf`, 2017.

[57] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[58] M. T. Jones and P. E. Plassmann, "A Parallel Graph Coloring Heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 654–669, May 1993.

[59] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating Graph Coloring on GPUs," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[60] D. W. Matula and L. L. Beck, "Smallest-last Ordering and Clustering and Graph Coloring Algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, Jul. 1983.

[61] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," in *Proceedings of the International Conference on World Wide Web (WWW)*, 1998.

[62] H. Yoon and G. S. Sohi, "Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[63] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. Hwu, "Automatic Execution of single-GPU Computations Across Multiple GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[64] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[65] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[66] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[67] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, 9:1–9:27, Aug. 2015.

[68] D. Grewe and M. F. P. O'Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC/ETAPS)*, 2011.

[69] T. Ramashekar and U. Bondhugula, "Automatic Data Allocation and Buffer Management for multi-GPU Machines," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, 60:1–60:26, Dec. 2013.

[70] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, 35:1–35:25, Dec. 2016.

[71] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-GPU System Design with Memory Networks," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[72] J. H. Zurawski, J. E. Murray, and P. J. Lemmon, "The Design and Verification of the AlphaStation 600 5-series Workstation," *Digital Tech. J.*, vol. 7, no. 1, pp. 89–99, Jan. 1995.

[73] *High Performance Fortran Language Specification*, 1997.

[74] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[75] T. Thanh-Hoang, A. Shambayati, and A. A. Chien, "A Data Layout Transformation (DLT) Accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2016.

[76] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[77] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1999.

[78] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[79] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): Chips that Remember and Compute," in *IEEE International Solids-State Circuits Conference (ISSCC)*, 1997.

[80] R. C. Murphy, P. M. Kogge, and A. Rodrigues, "The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems," in *Revised Papers from the Second International Workshop on Intelligent Memory Systems (IMS)*, 2000.

[81] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.

[82] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *IEEE International Conference on Computer Design (ICCD)*, 1999.

[83] D. Elliott, W. M. Snelgrove, and M. Stumm, "Computational Ram: A Memory-SIMD Hybrid and its Application to DSP," in *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, 1992.

[84] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Design Test of Computers*, vol. 16, no. 1, pp. 32–41, 1999.

[85] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[86] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[87] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-stacked DRAM," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[88] M. Chu, N. Jayasena, D. Zhang, and M. Ignatowski, "High-level Programming Model Abstractions for Processing in Memory," in *WoNDP: 1st Workshop on Near-Data Processing*, 2013.

[89] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *IEEE*

International Symposium on High Performance Computer Architecture (HPCA), 2017.

[90] R. Nair, S. Antao, C. Bertolli, P. Bose, J. Brunheroto, T. Chen, C.-Y. Cher, C. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, D. Gallo, L. Grinberg, J. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, K. O'Brien, M. Ohmacht, Y. Park, D. Prener, B. Rosenburg, K. Ryu, O. Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Z. Sura, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, 2015.

[91] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, 48:1–48:25, Dec. 2017.

[92] L. Nai, R. Hadidi, H. Xiao, H. Kim, J. Sim, and H. Kim, "CoolPIM: Thermal-Aware Source Throttling for Efficient PIM Instruction Offloading," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[93] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the Characteristics of 3D-Stacked Memories: A Case Study for Hybrid Memory Cube," *CoRR*, vol. abs/1706.02725, 2017. arXiv: 1706.02725.

[94] R. Hadidi, B. Asgari, J. Young, B. A. Mudassar, K. Garg, T. Krishna, and H. Kim, "Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018.

[95] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[96] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[97] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[98] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[99] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[100] ——, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[101] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, 2:1–2:38, Apr. 2013.

[102] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[103] C. McCurdy, A. L. Coxa, and J. Vetter, "Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[104] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering Translation Bandwidth with Virtual Caching," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[105] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling Page Table Walks for Irregular GPU Applications," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[106] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[107] J. Lee, M. Samadi, and S. Mahlke, "VAST: The Illusion of a Large Memory Space for GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[108] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[109] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[110] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping," in *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2015.